

*The Unabridged
Pentium[®] 4*

IA32 Processor Genealogy

First Edition

CD Addendum

*The Unabridged
Pentium® 4
IA32 Processor Genealogy
First Edition*

MINDSHARE, INC.

TOM SHANLEY

*TECHNICAL EDIT BY
BOB COLWELL*

ADDISON-WESLEY DEVELOPER'S PRESS

Reading, Massachusetts • Harlow, England • Menlo Park, California

New York • Don Mills, Ontario • Sydney

Bonn • Tokyo • Amsterdam • Mexico City • Seoul

San Juan • Madrid • Singapore • Paris • Taipei • Milan

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designators appear in this book, and Addison-Wesley was aware of the trademark claim, the designations have been printed in initial capital letters or all capital letters.

The authors and publishers have taken care in preparation of this book but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Library of Congress Cataloging-in-Publication Data

ISBN: 0-321-24656-X

Copyright ©2004 by MindShare, Inc.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher. Printed in the United States of America. Published simultaneously in Canada.

Sponsoring Editor:

Project Manager:

Cover Design:

Set in 10 point Palatino by MindShare, Inc.

1 2 3 4 5 6 7 8 9-MA-999897

First Printing, July 2004

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government, and Special Sales Department at (800) 238-9682.

Find A-W Developers Press on the World Wide Web at:
<http://www.aw.com/devpress/>

Chapter 1: Processor Hardware Feature Evolution

Introduction.....	13
In the Beginning, Processors and Memory Were Slow	13
Problem: Execution Unit Starvation	14
Problem Description	14
Solution: Instruction Prefetcher	14
Problem: Increasing Processor's Core Speed	15
Problem Description	15
Solution: Increase Frequency of Processor Input Clock	15
Problem: Narrow Data Bus.....	16
Problem Description	16
Solution: Increase Width of FSB Data Path	16
Problem: Increasing Processor Core Speed (Revisited).....	17
Problem Description	17
Solution: Multiply Input Clock Frequency On-Chip	18
Problem: Fast Processor, Slow Memory	18
Problem Description	18
An Impractical Solution	19
A Better Solution	19
An Additional Bonus.....	19
Problem: Processor Core Throttled by Slow FSB	20
Problem Description	20
Solution: Move External Cache On-Chip	21
Problem: FSB Protocol Inefficient When Fetching a Cache Line	22
Problem Description	22
Solution.....	24
Introduction	24
Non-Interleaved Memory Architecture.....	24
Two-Way Interleaved Memory Architecture	26
Four-Way Interleaved Memory Architecture.....	28
Problem: Separate FPU Is Slow	30
Problem Description	30
Solution: Move FPU On-Chip	30
Problem: Memory Writes Are Slow.....	32
Problem Description	32
Solution: On-Chip Posted Write Buffer	32
Problem: Internal Cache Contention.....	32
Problem Description	32
Solution: Separate Data and Code Caches	33
Problem: Single Instruction Pipeline Becomes Choke Point	34
Problem Description	34

Contents

Solution: Add Multiple Instruction Pipelines.....	34
Problem: Internal Cache Is Rather Small	34
Problem Description.....	34
Solution: Add External L2 Cache (first seen in 486 systems)	35
Problem: Prefetcher Can't Handle Branches.....	36
Problem Description.....	36
Solution: Add Branch Prediction Logic	37
Problem: Single Processor Can Become a Choke Point.....	39
Problem Description.....	39
Traditional Multiprocessing.....	39
CMP	40
Hyper-Threading	40
Problem: The x86 Instruction Set Is Complex	41
Problem Description.....	41
Solution: Go Primitive!.....	41
Problem: In-Order Execution Can Slow You Down.....	41
Problem Description.....	41
Solution: Out-of-Order Execution	42
Problem: L2 Access Speed Bound by FSB Speed.....	42
Problem Description.....	42
Solution: Move the L2 Closer to the Processor Core.....	43
Pentium® Pro.....	43
Pentium® II	44
Solution: Move the L2 Onto the Processor Die.....	44
Problem: Math Units Ill-Suited for Array Operations.....	46
Problem Description.....	46
Solutions.....	47
MMX Instruction Set and Registers	47
SSE Instructions and Registers.....	49
SSE2.....	51
SSE3.....	51
Problem: Execution Speed of Simple Math Operations.....	51
Problem Description.....	51
Solution: Double-Clock Simple ALUs.....	51
Problem: L1 Code Cache Is Inefficient	51
Problem Description.....	51
Solution: Execution Trace Cache.....	52
Problem: Some Applications Deal With Massive Databases.....	53
Problem Description.....	53
Solution: Add External L3 Cache.....	53
Solution: Move L3 Cache On-Chip.....	53

Problem: FSB Not Optimized for Multiprocessing.....	54
Problem Description.....	54
Solution: FSB Redesign.....	55
Problem: The IA Register Set Is Small	56
Problem Description.....	56
Solution: Instruction Register References Are Mapped To a Large Physical Register Set.....	57
Problem: Cache Miss Causes Cessation of Lookups.....	58
Problem Description.....	58
Solution: Non-Blocking Caches	59
Problem: Interrupt Delivery in Multiprocessor Environment.....	60
Problem Description.....	60
Solution: APIC.....	61
APIC Evolves in the Pentium® 4	63
Problem: No Data Prefetching.....	64
Problem Description.....	64
Solution: Add Data Prefetcher	64

Chapter 2: Register Set Evolution

100,000 Foot View	65
Introduction.....	65
Don't Forget the Acronym Glossary.....	65
Some Commonly-Used Terms.....	66
The Purpose of the Register Set	67
Control Registers.....	67
Key OS Data Structure Pointer Registers	67
Instruction Pointer Register Pair.....	68
Debug and Test Registers	68
General Purpose Data Registers	68
Status Registers.....	68
FPU Register Set	69
Error Logging Registers	69
Interrupt-Related Registers	69
Performance Monitoring Registers.....	69
Time Stamp Counter (TSC) Register	69
Memory Type and Range Registers (MTRRs)	69
Register Set Evolution.....	70
The 386 Register Set.....	77
Control Registers.....	77
CR0.....	77
CR1.....	79
CR2.....	79

Contents

CR3.....	79
EFlags Register	80
General Purpose Registers (GPRs)	84
Introduction	84
EAX, EBX, ECX and EDX Registers	85
EBP Register.....	85
Index Registers	86
Segment Registers	87
Real Mode Usage	87
Protected Mode Usage	87
DS, ES, FS and GS	88
CS	88
SS	88
Extended Instruction Pointer (EIP) Register	90
Task Register.....	90
What Is a TSS?	90
The Purpose of the Task Register	91
GDTR and LDTR.....	93
IDTR.....	95
Hardware Interrupts	95
Software Exceptions and Interrupts.....	95
IDTR Points To the Interrupt Table.....	96
The Debug Registers.....	97
Test Registers	99
The 486 Register Set.....	100
FPU Register Set	101
New Test Registers	101
CR0 Updated	102
CR3 Updated	103
CR4 Added.....	103
EFlags Register Updated.....	104
The Pentium® Register Set.....	105
CR4 Added.....	108
MMX Register Set Added	109
Local APIC Added.....	109
Model Specific Registers (MSRs) Added	110
Machine Check Architecture Registers Added	112
Performance Monitoring Registers Added	112
Time Stamp Counter (TSC) Added	113
DR7 Enhanced	113
Test Registers Deleted	113

The Pentium® Pro Register Set.....	113
CR4 Bits Added	115
Additions to the Local APIC Register Set.....	116
Added Performance Counter Overflow Interrupt.....	116
APIC's Base Address Assignable	116
MMX Not Implemented.....	116
New MSRs.....	116
The Pentium® II Register Set.....	117
MMX Added Back In.....	118
Pentium® II and III MSR Changes.....	118
The Pentium® III Register Set	118
New CR4 Bits.....	120
SSE Register Set Added.....	121
IA32_Platform_ID MSR Added	121
The Pentium® 4 and Pentium® M Register Set	121
FP Opcode Register Added	123
DAZ Bit Added to MXCSR.....	123
Local APIC Changes.....	123
MSR Register Set Changes.....	125

Chapter 3: Instruction Set Evolution

Introduction.....	127
386 Instruction Set.....	127
486 Additions to the Instruction Set.....	134
Pentium® Additions to the Instruction Set.....	136
Pentium® Pro Additions to the Instruction Set.....	139
Pentium® II Additions to the Instruction Set.....	141
Pentium® III Additions to the Instruction Set	143
Pentium® 4 Additions to the Instruction Set.....	145

Chapter 4: Exception Evolution

Introduction.....	149
Exception Evolution.....	149
Exceptions Defined For the 386.....	149
486 Exception Additions/Changes	152
Pentium® Exception Additions/Changes.....	153
Pentium® Pro and Pentium® II Exception Additions/Changes	154
Pentium® III	154
Pentium® 4/Pentium® M Exception Additions/Changes.....	154

Contents

Chapter 5: P6 Road Map

Introduction.....	155
The P6 Processor Family	155
The Klamath Core	155
Pentium® Pro Processor.....	156
Dual-Cavity PGA Package	156
Dedicated L2 Cache	157
BSB Speed and Width.....	157
L1 Data Cache.....	158
L1 Code Cache.....	158
FSB Speed.....	158
FSB Protocol.....	159
FSB Arbitration Scheme	159
Address Bus Width.....	159
PAE-36 Mode.....	159
No MMX.....	160
Power Conservation States.....	160
AutoHalt Power Down State	160
Stop Grant State	161
Halt/Grant Snoop State.....	162
Dual-Independent Bus Architecture (DIBA)	162
Early Pentium® II Processor.....	163
Processor Cartridge and Slot.....	163
The L2 Cache	164
The BSB Speed and Width.....	165
The L1 Data Cache.....	165
The L1 Code Cache	165
The FSB Speed	165
The FSB Arbitration Scheme	165
Address Bus Width.....	165
Power Conservation States.....	166
Sleep State	166
Deep Sleep State.....	167
MMX	167
Enhanced Legacy Code Execution Speed	168
Pentium® II Overdrive Processor	171
The Deschutes Core	171
Later Pentium® II Processor	172
Celeron Covington Processor.....	173
Celeron Mendocino Processor	173
Pentium® II Xeon Processor	174

Celeron Dixon Processor	174
The Katmai Core.....	175
Early Pentium® III Processor.....	176
Pentium® III Coppermine Processor.....	176
Xeon Tanner Processor.....	177
Xeon Cascades Processor	177

Chapter 6: Overview of the P6 Core

Description of the P6 Processor Core	179
Introduction to the Internal Architecture	179
The FSB Interface Unit.....	179
IO Read or Write Transaction	180
Memory Read Transaction	180
Memory Write Transaction	180
Memory Read and Invalidate Transaction.....	181
Special Transaction	182
Interrupt Acknowledge Transaction.....	182
Branch Trace Message Transaction	182
Deferred Reply Transaction	182
The Backside Bus (BSB) Interface Unit.....	183
The Unified L2 Cache	183
The L1 Data Cache	183
The L1 Code Cache	183
The Processor Core	183
The Local APIC Unit.....	184
The IA32 Instruction Set Is Variable Length	185
IA32 Instructions Are Converted Into μops	186
The IA32 Data Register Set	186
The Data Register Set Is Small.....	186
Elimination of False Dependencies	187
Introduction to the Instruction Pipeline.....	190
Instructions Are Executed Out of Order.....	193
The Branch Prediction Mechanisms.....	193
486 Branch Handling	193
Pentium® Branch Prediction	193
P6 Branch Prediction	194
Mispredicted Branches are VERY Costly!	194
Dynamic Branch Prediction	196
General	196
Yeh's Prediction Algorithm.....	196
Return Stack Buffer (RSB).....	197
Static Branch Prediction	199

Contents

Code Optimizations.....	200
General.....	200
Reduce the Number of Branches	200
Follow the Static Branch Prediction Algorithm.....	201
Identify and Improve Unpredictable Branches	201
Don't Intermingle Code and Data	201
Align Data	201
Avoid Serializing Instructions	201
Where Possible, Do Context Switches in Software	202
Eliminate Partial Stalls: Small Write Followed by Full-Register Read.....	202
Data Segment Register Changes Serialize Execution	203
BSP Selection Process.....	203

Chapter 7: Overview of the P6 FSB Basics

Note.....	205
Agents.....	205
Agent Types	205
Multiple Personalities.....	206
Uniprocessor vs. Multiprocessor Bus.....	206
Request Agents.....	208
The Request Agent Types	208
The Agent ID	208
What the Agent ID Is Used For.....	208
How the Agent ID Is Assigned	209
The Transaction Phases.....	209
The Pentium® Transaction Phases	209
The P6 FSB Transaction Phases.....	209
Transaction Pipelining.....	210
The FSB Is Divided into Signal Groups	210
Step 1: Gain Ownership of the Request Phase Signal Group	210
Step 2: Issue the Transaction Request	211
Step 3: Yield Request Phase Signal Group, Proceed to Next Group	211
Phases Proceed in Predefined Order.....	211
The Request Phase	212
The Error Phase	212
The Snoop Phase	212
The Response Phase	213
The Data Phase(s)	213
Next Agent Can't Use Signal Group Until Current Agent Done With It	214
Transaction Tracking.....	215
Request Agent Transaction Tracking.....	215
Snoop Agent Transaction Tracking.....	215

Response Agent Transaction Tracking	216
The In Order Queue (IOQ)	216
Pentium® 4/Pentium® M Differences.....	216

Chapter 8: P6 FSB Electrical Overview

Applicability to the Pentium® 4/Pentium® M	217
Different Voltage Levels	217
AGTL+	217
Introduction.....	218
Everything's Relative	218
All Signals Are Asserted When Low	221
Pull-ups Return Lines Back to an Electrical High	222
The Layout.....	222
Synchronous Bus.....	223
How High is High and How Low is Low?	224
After Sampling, Minimum Response Time Is One Clock.....	225

Chapter 9: P6 FSB Arbitration Basics

Applicability to the Pentium® 4/Pentium® M	227
Detailed Description of FSB Arbitration.....	227
What Signal Group are You Arbitrating For?.....	227
Symmetric Agent Arbitration—Democracy at Work	228
Introduction	228
No External Arbiter Required	228
Agent ID Assignment.....	230
Arbitration Algorithm	230
Rotating ID.....	230
Busy / Idle State.....	232
Bus Parking.....	233
Be Fair!	233
Requesting Ownership.....	233
Detailed Description of P6 Symmetric Arbitration	235
Priority Agent Arbitration.....	235
Request to Break Into the Rotation.....	235
Example Priority Agents.....	236
BPRI# Is Not a Shared Signal	236
Priority Agent Parking.....	237
Be Fair to the Common People.....	237
Detailed Description of P6 Priority Agent Arbitration.....	237
Locking—Shared Resource Acquisition.....	238
Introduction	238
Shared Resource Concept	238

Contents

Testing Availability and Gaining Ownership of a Shared Resource.....	239
Race Condition Can Present Problem.....	239
Guaranteeing Atomicity of Read / Modify / Write.....	240
Detailed Description of P6 FSB Locking.....	241
Blocking New Requests—Stop! I’m Full!.....	241
Introduction.....	241
BNR# is Shared Signal.....	242
Detailed Description of P6 Transaction Blocking.....	242

Chapter 10: P6 FSB Request & Error Phase Basics

Applicability to the Pentium® 4/Pentium® M.....	245
The Pentium® 4 Request Phase Is Double-Pumped.....	245
Request Phase Parity Scheme Was Altered.....	246
The Error Phase Was Eliminated in the Pentium® 4.....	246
Introduction to the Request Phase.....	246
A Request Consists of a Large Amount of Information.....	246
Introduction to the Request Phase Signal Group.....	248
Request Phase Signal Group is Multiplexed.....	249
Packet A Content.....	250
Packet B Content.....	250
Introduction to the Transaction Types.....	252
Detailed Description of the P6 Request Phase.....	254
Introduction to the Error Phase.....	255
The Error Phase Purpose: Check for Good Receipt.....	255
Pentium® 4 FSB Protocol Eliminated the Error Phase.....	256

Chapter 11: P6 FSB Snoop Phase Overview

Applicability to the Pentium® 4/Pentium® M.....	257
The Snoop Phase Has Two Purposes.....	257
The Snoop Phase Signals.....	258
Introduction.....	258
The Snoop Result Signals are Shared, DEFER# Isn’t.....	258
The Agents Involved in the Snoop Phase.....	258
The Request Agent’s Role in the Snoop Phase.....	259
The Snoop Agent’s Role in the Snoop Phase.....	259
The Response Agent’s Role in the Snoop Phase.....	259
The Duration of the Snoop Phase Is Variable.....	260
Is There a Snoop Stall Duration Limit?.....	263
Memory Transaction Snooping.....	263
Non-Memory Transactions Have a Snoop Phase.....	267

Chapter 12: P6 FSB Response & Data Phase Basics

Applicability to the Pentium® 4/Pentium® M	269
The Response Phase.....	269
The Data Phase	269
Response Phase Basics	270
The Response Phase Start Point	270
The Purpose of the Response Phase	270
The Response Phase Signal Group	271
The Response Phase End Point	272
The List of Responses	272
The Response Phase May Complete a Transaction.....	274
Detailed Description of the Response Phase.....	274
Data Phase Basics	275
The P6 Data Phase Signal Group	275
Detailed Description of the Data Phase	275

Chapter 13: P6 FSB Deferral Basics

Applicability to the Pentium® 4/Pentium® M	277
Permission to Defer Transaction Completion	277
DEFER# Assertion Delays Transaction Completion	278
Mail Delivery Analogy	278
Example System Operation Overview	278
The Wrong Way: Stall Until You Have Completion.....	279
The Right Way: Break the Connection and Reply Later	279
Root Complex Should be a Faithful Messenger	280
What If HITM# and DEFER# Are Both Asserted?	281
Detailed Deferred Transaction Description	282

Chapter 14: P6 FSB Central Agent Transactions

Applicability to the Pentium® 4/Pentium® M	283
Point-to-Point vs. Broadcast	283
No Address Is Output	285
Interrupt Acknowledge Transaction	285
Background.....	285
How the P6 Processors Differed from Earlier IA32 Processors.....	287
Special Transaction	287
General.....	287
The Message Is Encoded on the Byte Enables	287
The Message Types	288

Contents

The BTM Transaction Aids in Program Debug	290
The Problem.....	290
The Solution.....	290
Enabling BTM Capability	291
Branch Trace Message Transaction	292
Proper Response to a BTM Transaction	292
Detailed Description of the BTM Transaction	292

Chapter 15: Self-Modifying Code and the P6

Introduction.....	293
Description	293
Don't Let Your Data and Code Get Too Friendly!	295

Chapter 16: Pentium® 4 FSB Screenshots

Courtesy of Agilent Technologies	321
Example 1.....	322
Example 2.....	323
Example 3.....	324
Example 4.....	325
Example 5.....	326
Example 6.....	327

Index	317
--------------------	------------

1-1	External Clock Was Used as Core Speed and FSB Speed	4
1-2	Clock Multiplier	6
1-3	External Cache.....	8
1-4	486 Processor Block Diagram	9
1-5	386/486 Address and Data Buses.....	11
1-6	Burst Cache Line Fill Operation	12
1-7	Non-Interleaved Memory Architecture.....	13
1-8	Two-Way Interleaved Memory	16
1-9	Four-Way Interleaved Memory	17
1-10	486 Block Diagram.....	19
1-11	Pentium® Processor Block Diagram.....	21
1-12	Add an External L2 Cache	24
1-13	Pentium® Added Branch Prediction Logic	26
1-14	A System Incorporating Multiple Processor Chips	28
1-15	Add an External L2 Cache	31
1-16	Pentium® Pro Packaging.....	33
1-17	Pentium® II Cartridge.....	33
1-19	Pentium® II Cartridge: View 2	34
1-18	Pentium® II Cartridge: View 1	34
1-20	Pentium® Register Set Includes a Set of 64-bit MMX Data Registers.....	36
1-21	Pentium® III Register Set	38
1-22	The Pentium® 4 Added the Trace Cache	40
1-23	Off-Chip and On-Chip L3 Cache.....	42
1-24	P6 FSB Transaction Pipelining	44
1-25	IA Data Registers	46
1-26	Blocking Cache	47
1-27	Non-Blocking Cache.....	48
1-28	Old-Style Interrupt Delivery	49
1-29	Distributed Interrupt Delivery.....	50
1-30	The Pentium® 4 Improved the Interrupt Delivery Mechanism	51
1-31	The Pentium® 4 Added a Data Prefetcher.....	52
2-1	386 Register Set.....	58
2-2	486 Register Set.....	59
2-3	Pentium® Register Set	60
2-4	Pentium® Pro Register Set	61
2-5	Pentium® II Register Set.....	62
2-6	Pentium® III Register Set	63
2-7	Pentium® 4 and Pentium® M Register Set.....	64
2-8	386 Register Set.....	65
2-9	386 Control Register 0 (CR0)	67
2-10	Control Register 2 (CR2)	68
2-11	386 Control Register 3 (CR3)	68

Figures

2-12	386 Extended Flags Register (EFlags)	69
2-13	The General Purpose Registers (GPRs)	75
2-14	The Segment Registers	76
2-15	Segment Registers Identify Memory Areas Associated with Executing Program.....	77
2-16	EIP Register.....	78
2-17	Task State Segment (TSS).....	80
2-18	The Task Register Identifies the TSS Associated with the Currently Executing Task	81
2-19	GDTR and LDTR.....	82
2-20	The IDT in Real Mode	84
2-21	The IDT in Protected Mode	85
2-22	Debug Registers	86
2-23	386 Test Registers.....	87
2-24	486 Register Set.....	88
2-25	CR0 in the 486.....	90
2-26	CR3 in the 486.....	91
2-27	CR4 in the 486.....	92
2-28	EFlags Register	93
2-29	Pentium® Register Set	95
2-30	CR4 in the Pentium®.....	97
2-31	Pentium® Pro Register Set	102
2-32	CR4 in the Pentium® Pro.....	103
2-33	Pentium® II Register Set.....	106
2-34	Pentium® III Register Set	108
2-35	Pentium® 4 and Pentium® M Register Set.....	111
2-36	Thermal Sensor Interrupt Entry Added to the LVT	113
2-37	APIC ID Register Format Changed.....	113
3-1	Pentium® Capabilities	126
3-2	Pentium® Pro Capabilities	129
3-3	Pentium® II Capabilities.....	131
3-4	Pentium® III Capabilities	133
3-5	Pentium® 4/Pentium® M Capabilities	135
5-1	P6 Klamath Core Roadmap	144
5-2	Pentium® Pro Dual-Cavity PGA Package	151
5-3	SEC Package (1-of-2).....	157
5-4	SEC Package (2-of-2).....	157
5-5	SEC Cartridge Cover	158
5-6	Pentium® II Block Diagram (Simplified)	158
5-7	P6 Deschutes Core Roadmap	160
5-8	P6 Katmai Core Roadmap	163
6-1	P6 Overview	173

6-2	IA32 Data Registers	177
6-3	The ROB	178
6-4	Instruction Pipeline	178
6-5	A Branch Mispredict Is VERY Costly! Causes Entire Pipeline to Be Flushed..	183
6-6	Branch Target Buffer (BTB)	185
6-7	Return Stack Buffer (RSB).....	186
6-8	Static Branch Prediction Algorithm	187
6-9	Static Branch Prediction Logic	188
7-1	Block Diagram of a Typical System	195
7-2	Bus Signal Groups	198
7-3	Pipelined Transactions.....	202
8-1	A GTL+ Input Comparator	207
8-2	Voltage Divider Sources Reference Voltage to Device's Input Comparators..	207
8-3	The Pull-up Voltage (V_{tt}) = 1.5Vdc.....	208
8-4	An Electrical Low = a Logical High	208
8-5	An Electrical High = a Logical Low	209
8-6	There Are Pull-ups at Both Ends of Each Trace	210
8-7	Maximum Stub Length	211
8-8	Inter-Device Spacing	211
8-9	Electrical High Limits.....	212
8-10	Electrical Low Limits.....	213
8-11	Example.....	214
9-1	System Block Diagram (Single Processor Cluster).....	217
9-2	System Block Diagram (Multiple Processor Clusters).....	218
9-3	Reset Initializes the Rotating ID	219
9-4	Ownership Passes from CPU to CPU in a Strict Rotation	220
9-5	The Busy/Idle Indicator	221
9-6	Quad-Processor Pentium® Pro System	222
9-7	Bus Request Signal Interconnect.....	223
9-8	Dual Cluster, Eight-Way System Block Diagram.....	225
9-9	All FSB Agents Must Track FSB Activity	230
9-10	The In Order Queue	231
9-11	The In Order Queue and BNR#	231
9-12	All Agents Capable of Initiating Transactions Must Monitor BNR#	232
10-1	A Transaction Request Consists of a Large Amount of Information	236
10-2	Two Information Packets Broadcast during Request Phase.....	237
10-3	Packet A Signal Names	239
10-4	Packet B Signal Names.....	240
10-5	Error Phase.....	243
11-1	System Block Diagram	248
12-1	Actions Taken on a Memory Write That Hits on a Modified Line.....	259
13-1	System with Four Processors and Two FSB/PCI Bridges	269

Figures

13-2	Two Processor Clusters Linked by Cluster Bridge	270
14-1	Example PCI Express System.....	272
14-2	An External, Hardware Interrupt Delivered to the Processor's INTR Pin.....	274
14-3	The Message Is Encoded on the Processor's Byte Enables	275
14-4	A Cache Can Hide Program Execution from an External FSB Analyzer	279
14-5	DEBUGCTL MSR.....	279
15-6	The Assumption.....	284
15-7	The Code Line Is Requested.....	285
15-8	The L1 Code Cache Miss.....	286
15-9	The Code Request Is Forwarded to the L2 Cache	287
15-10	The Code Request Is Forwarded to the FSB.....	288
15-11	The Line Is Fetched from Memory	289
15-12	The Line Is Placed in the L2 Cache and the L1 Data Cache.....	290
15-13	The Line Is Fed Into the Pipeline	291
15-14	A Load Is Executed to Read an Instruction from Memory into a Register	292
15-15	The Line Is in the L2 Cache	293
15-16	The Line Is Forwarded to the L1 Data Cache and to the Register	294
15-17	The Instruction Is Altered in the Register	295
15-18	The Modified Instruction is Stored Back to Memory	296
15-19	The Triple Lookup	297
15-20	The Pipeline Is Blown.....	298
15-21	The L1 Data Cache Line Is Updated	299
15-22	The Modified Line Is Forwarded to the FSB Interface Unit	300
15-23	The Line Is Invalidated in the L1 Data Cache.....	301
15-24	The Next Time the Line Is Fetched, It Results In a Cache Miss	302
15-25	The Line Read Request Is Forwarded to the FSB Interface Unit.....	303
15-26	The Memory Code Read Transaction Is Performed on the FSB	304
15-27	The Self-Snoop Results in a Hit on the Modified Line in the Writeback Buffer.....	305
15-28	The FSB Interface Unit Provides the Modified Line to Memory and to the CPU.....	306
15-29	The Line Is Fed into the Pipeline to Be Executed	307
15-30	Don't Let Your Code and Data Get Too Friendly	308
16-1	Example 1.....	310
16-2	Example 2.....	311
16-3	Example 3.....	312
16-4	Example 4.....	313
16-5	Example 5.....	314
16-6	Example 6.....	315

1-1	Prefetch Queue Depth.....	2
1-2	Evolution of the Processor's Data Bus Width.....	5
1-3	Cache Line Fill Toggle Mode Transfer Sequence	15
2-1	CR0 in the 386.....	66
2-2	386 EFlags Register	69
2-3	Pentium® MSRs	98
2-4	Pentium® III Added New Bits in CR4.....	109
3-1	386 Instruction Set.....	116
3-2	486 Instruction Set Additions.....	123
3-3	Pentium® Instruction Set Additions/Changes	125
3-4	Pentium® Pro Instruction Set Additions/Changes.....	127
3-5	Pentium® II Instruction Set Additions/Changes	130
3-6	Pentium® III Instruction Set Additions/Changes	132
3-7	Pentium® 4/Pentium® M Instruction Set Additions/Changes.....	134
4-1	386 Exceptions	138
4-2	New or Altered 486 Exceptions	141
4-3	New or Altered Pentium® Exceptions	141
4-4	New Pentium® III Exceptions	142
6-1	Pipeline Stages.....	179
10-1	Packet B Signal Names.....	238
10-2	Currently-Defined Transaction Types.....	241
11-1	Snoop Result Table	249
11-2	Effects of Snoop.....	252
12-1	Response List.....	260
14-1	Message Types	276

Tables

1

Processor Hardware Feature Evolution

Introduction

The processor fulfills a central role in the overall system and each processor feature addresses a specific, processor-related problem. Rather than plunging immediately into a detailed discussion of the latest Intel® IA32 processor, this chapter highlights a number of important problems faced by processor designers over the years. For each problem area, the respective processor feature(s) that addresses that problem is introduced.

A detailed discussion of each feature can be found in the chapters on each processor.

In the Beginning, Processors and Memory Were Slow

Way back when, everything was slow:

- Processors were clocked by a very slow clock and it took them a number of these long, wide clock cycles to execute just one instruction.
- The processor's FSB was equally slow in performing bus transactions.
- The amount of time it took to initiate and complete a memory access was quite long.

The Unabridged Pentium® 4

- When the IBM PC was introduced, it was based on the Intel® 8088 processor running at 4.77MHz. The 8088 took four ticks of its 4.77MHz clock to run a zero wait state bus transaction—a duration of 838ns ($4 \times 209\text{ns} = 838\text{ns}$). Virtually any DRAM or ROM on the market at that time could respond well within this period of time, so memory did not throttle processor performance. The bottom line was everything ran slowly.

and everyone was happy!

Problem: Execution Unit Starvation

Problem Description

The processor fetches an instruction from memory and dispatches it to the execution unit to be executed. If the processor were to wait until the current instruction completes execution before initiating a memory read transaction on its FSB to fetch the next instruction from memory, this would result in very poor performance.

Solution: Instruction Prefetcher

Refer to Table 1-1 on page 2. In an effort to decouple the processor's execution unit from the slowness of memory, Intel® incorporates an instruction prefetcher and a prefetch queue into each processor.

Table 1-1: Prefetch Queue Depth

Processor	Queue Depth (in bytes)
8088	4
8086	6
286	6
386	16
486	32

Chapter 1: Processor Hardware Feature Evolution

Table 1-1: Prefetch Queue Depth (Continued)

Processor	Queue Depth (in bytes)
Pentium®	Two 64-byte queues
Pentium® Pro and Pentium® II	32 bytes
Pentium® III	32 bytes
Pentium® 4 without HyperThreading	Two 64-byte queues
Pentium® 4 with HyperThreading	One 64-byte queue per logical processor

Problem: Increasing Processor's Core Speed

Problem Description

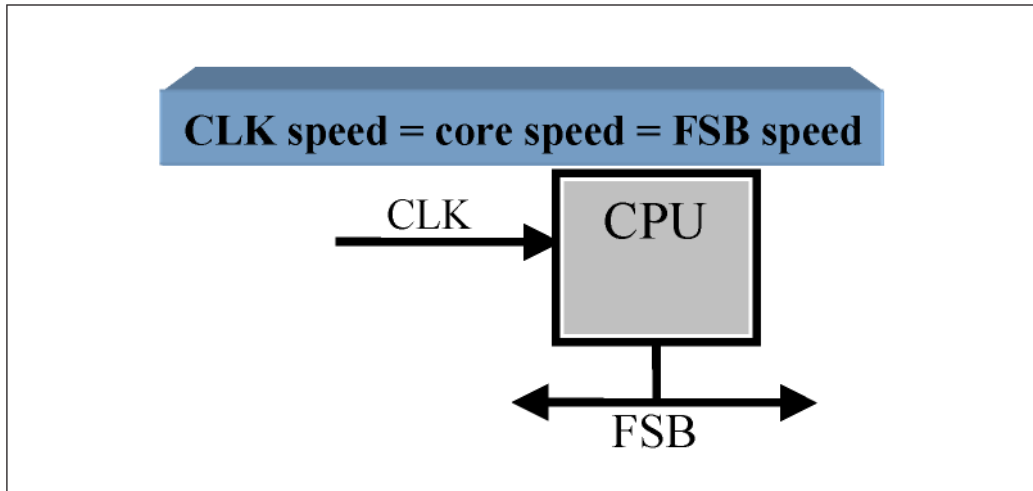
Refer to Figure 1-1 on page 4. Prior to the advent of the Pentium® processor, the clock signal supplied to the processor's CLK input was used to clock the processor's internal logic as well as defining the speed of the processor's FSB. The 8088 processor used in the first IBM PC was based on a 4.77MHz clock speed. It stands to reason that the frequency of the supplied external clock is one of the major factors defining the performance of the processor.

Solution: Increase Frequency of Processor Input Clock

Over the course of time, Intel® increased the frequency of the external clock supplied to the processor by the crystal oscillator on the system board. The clock frequency increase resulted in a processor core able to execute instructions more quickly. In addition, data could be transferred more quickly over the faster FSB. The last Intel® processor that used the external clock to directly clock the processor core logic was the 66MHz Pentium® processor.

The Unabridged Pentium® 4

Figure 1-1: External Clock Was Used as Core Speed and FSB Speed



Problem: Narrow Data Bus

Problem Description

The Intel® 8088 processor's FSB implemented one 8-bit data path, enabling it to read or write one byte per transaction on the FSB. This obviously throttled the performance of the processor in transferring information between itself and external devices (such as system memory and IO devices).

Solution: Increase Width of FSB Data Path

Refer to Table 1-2 on page 5. Over time, the width of the FSB data path was increased, thereby permitting more data to be transferred during each read or a write transaction performed on the FSB. However, it should be noted that everything in life involves tradeoffs. Increasing the data bus width also increases the pin count on the processor and the chipset, as well as increasing the number of traces that must be routed on the system board.

Chapter 1: Processor Hardware Feature Evolution

Table 1-2: Evolution of the Processor's Data Bus Width

Processor	Data Bus Width
8088	8 bits (one byte)
8086	16 bits (2 bytes)
286	
386SX	
386DX	32 bits (4 bytes)
486SX	16 bits (2 bytes)
486DX	32 bits (4 bytes)
Pentium®	64 bits (8 bytes)
Pentium® Pro	
Pentium® II	
Pentium® III	
Pentium® 4	
Pentium® M	

Problem: Increasing Processor Core Speed (Revisited)

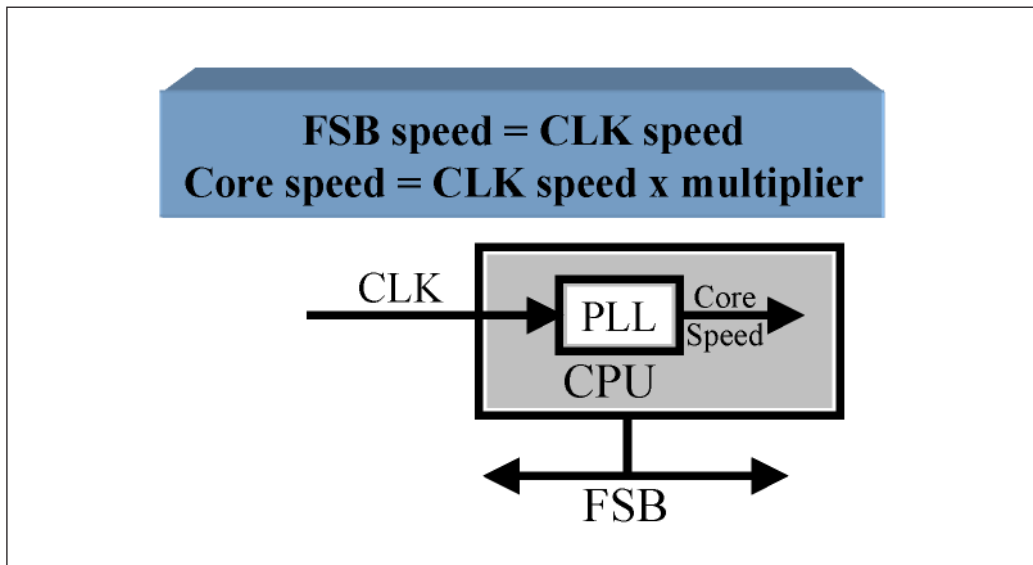
Problem Description

System board design is complicated when traces carrying high frequency signals must be routed. Problems associated with the high speed external clock supplied to the processor (as well the FSB-related signals) and the chipset became significant when laying out a system board with signal traces running at greater than 66MHz.

Solution: Multiply Input Clock Frequency On-Chip

Refer to Figure 1-2 on page 6. Starting with the 486SX2, 486DX2 and 486DX4 processors, Intel® included a Phase Lock Loop (PLL) on the processor chip to multiply the external clock and thereby achieve a higher frequency clock for use on the processor die. The FSB Interface Unit, however, ran directly off the external clock.

Figure 1-2: Clock Multiplier



Problem: Fast Processor, Slow Memory

Problem Description

A system may incorporate a processor that performs internal instruction execution at a high clock rate and that implements a very efficient, fast FSB. If such a system's memory is populated with memory modules with an access time that is significantly slower than the FSB speed, the memory's ability to source code and data to the processor will throttle the processor's performance.

Chapter 1: Processor Hardware Feature Evolution

An Impractical Solution

The system designer could populate system memory with high-speed static RAM. Although this would address the speed issue, it creates a number of other problems. The static RAM:

- is very expensive.
- has a higher pin/trace count than DRAM.
- consumes significantly more power than DRAM.
- generates significantly more heat than DRAM.

A Better Solution

Refer to Figure 1-3 on page 8. The system board designer could address the problem by placing an external cache between the processor and the system (including system memory). When the processor initiates a memory access on its FSB, the external cache performs a lookup and, in the event of a cache hit, can service the processor-initiated memory read or write from the fast static RAM that comprises the cache. In the event of a cache miss on a memory read, the external cache then performs the access to system memory, obtains the requested line, places a copy of the line in the external cache, and sources the line to the processor over the processor's FSB.

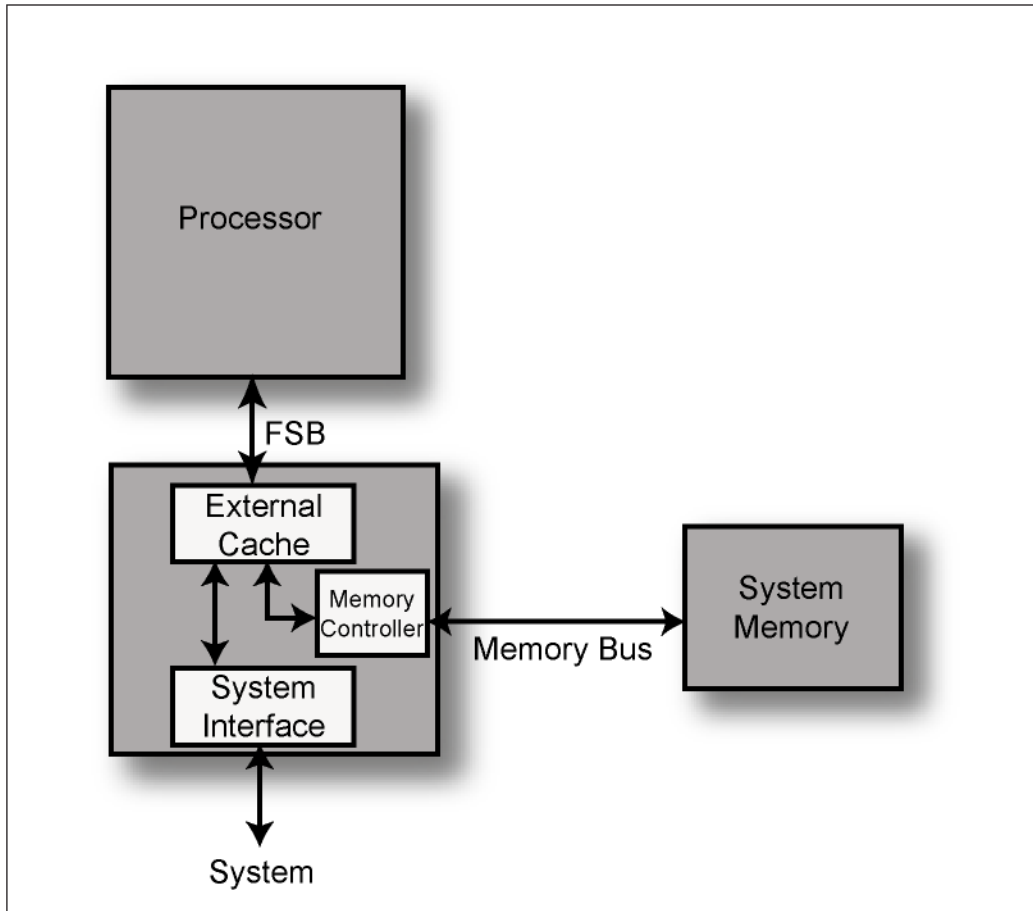
Over time, the external cache accumulates copies of the most frequently accessed code and data lines, thereby providing fast access for any subsequent processor-initiated accesses to code or data within those lines.

This solution was first implemented in 386-based systems, but can also be found in some of today's high-end workstations and servers.

An Additional Bonus

As an additional bonus, during periods of time wherein processor memory requests are fulfilled from the external cache, the system memory remains available for memory accesses initiated by other processors or by non-processor entities (e.g., hard disk controllers, network controllers, etc.).

Figure 1-3: External Cache

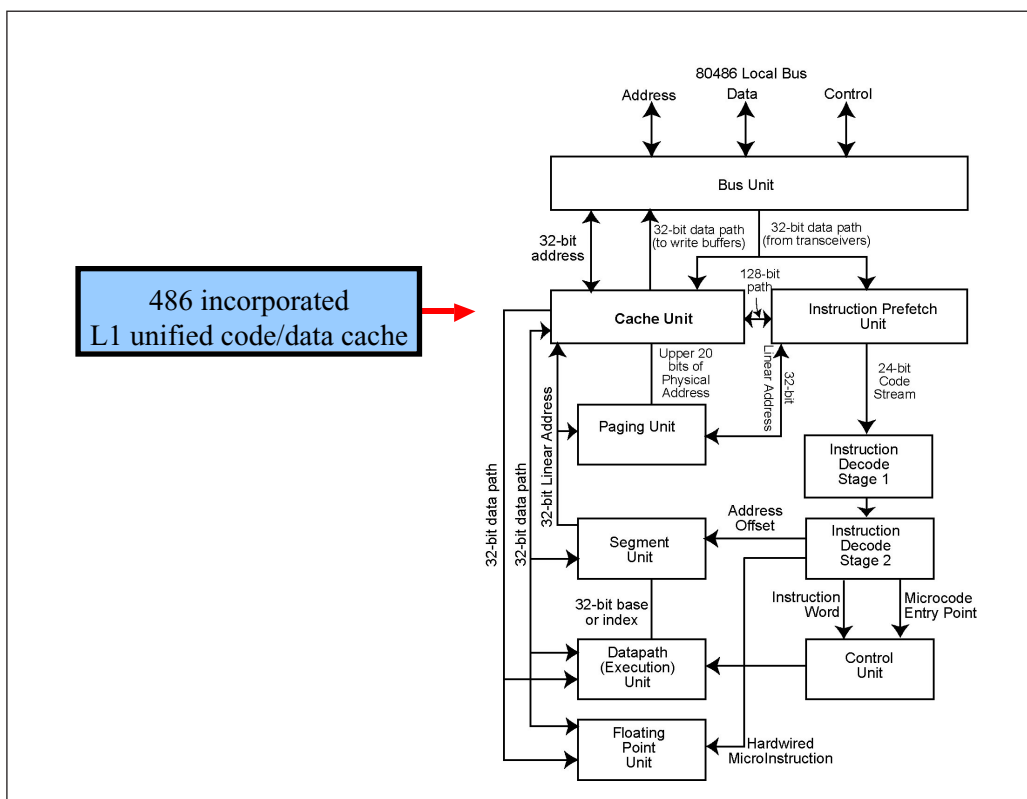


Problem: Processor Core Throttled by Slow FSB

Problem Description

Although the addition of an external cache results in better processor performance, as the processor's internal speed increases well beyond that of the FSB, the FSB can become a bottleneck.

Figure 1-4: 486 Processor Block Diagram



Problem: FSB Protocol Inefficient When Fetching a Cache Line

Problem Description

Prior to the 486, the processor's FSB protocol was not designed to efficiently handle expeditious cache line reads from memory (but it didn't matter, because there was no cache).

Starting with the 486, all IA32 processors incorporate at least one internal cache. When the 486 processor core must perform a code access (to prefetch the next instruction) or a data access, the code or data request is submitted to the internal cache to determine if the line that contains the requested information is currently in the cache. The 486 had a cache line size of 16 bytes. In the event of a cache miss, the processor initiates a memory code or data read transaction on its FSB to obtain the line that contains the requested code or data.

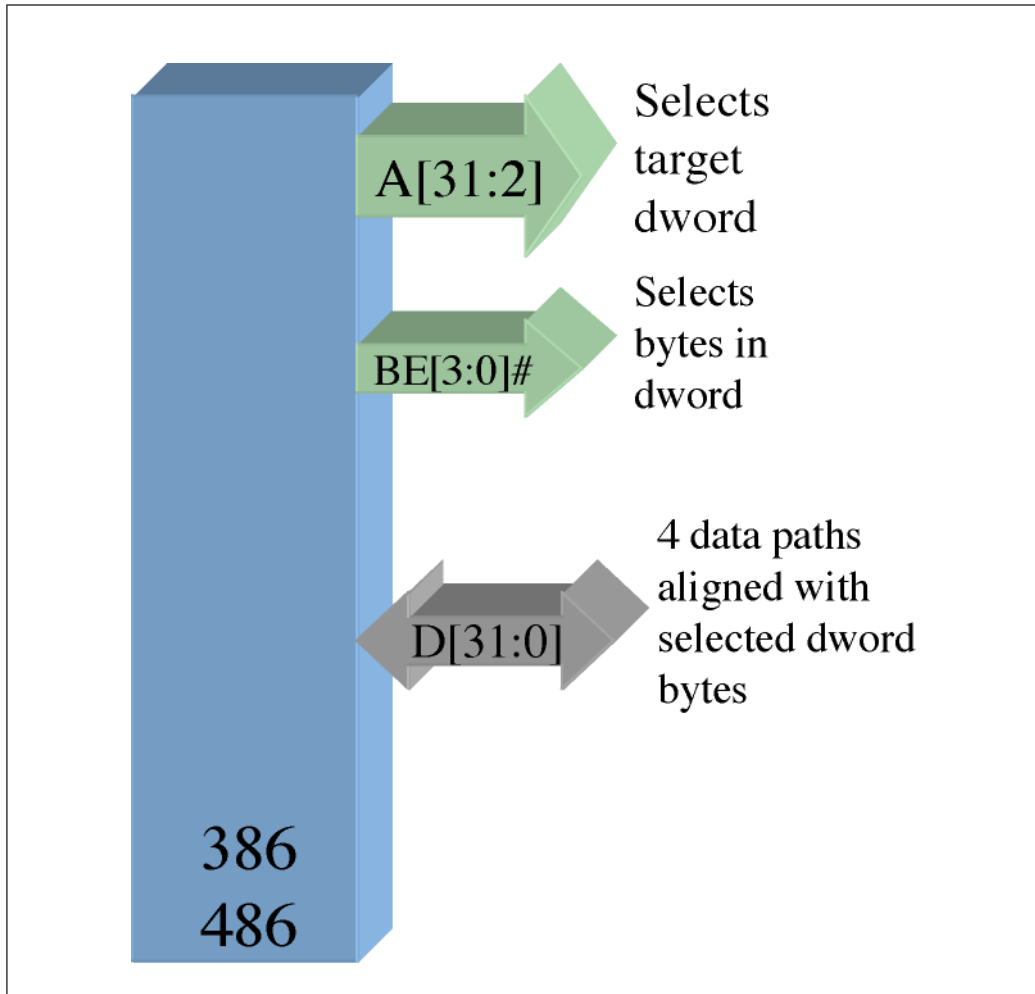
Refer to Figure 1-5 on page 11. A 16-byte line of information starts on an address divisible by 16d and consists of four dwords. If the 486 had been designed such that it used the same FSB protocol as the 386, it would have to initiate four separate memory read transactions to obtain each of the four dwords that comprise the requested line. To obtain each of the four dwords, it would have to:

- Arbitrate for ownership of the FSB.
- address the dword, and
- await delivery of the dword.

This would result in slow performance.

Chapter 1: Processor Hardware Feature Evolution

Figure 1-5: 386/486 Address and Data Buses



Solution

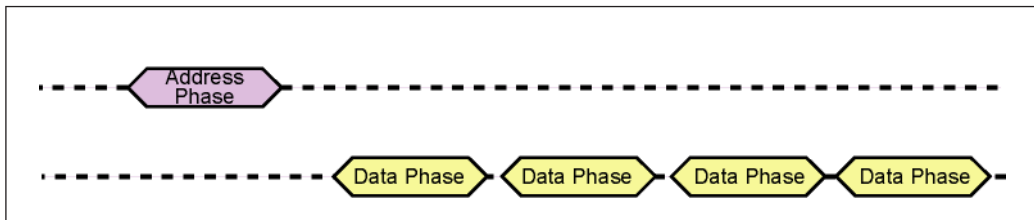
Introduction

Refer to Figure 1-6 on page 12. The FSB protocol for the 486 and all subsequent IA32 processors implements a burst cache line fill transaction consisting of:

- An Address Phase to deliver the memory address and transaction type to the memory controller.
- Four Data Phases, each of which reads or writes one fourth of a cache line (from or to memory).

The dword addressed in the Address Phase is referred to as the critical dword—i.e., the dword that contains the code or data that was originally requested by the processor core and which caused the cache miss. The critical dword could be any of the four dwords comprising the line.

Figure 1-6: Burst Cache Line Fill Operation



Non-Interleaved Memory Architecture

Refer to Figure 1-7 on page 13. Assume that the processor experienced a read miss on its internal cache and initiated a 16-byte memory read transaction on its FSB. The system memory controller would accomplish the line fetch in the following manner:

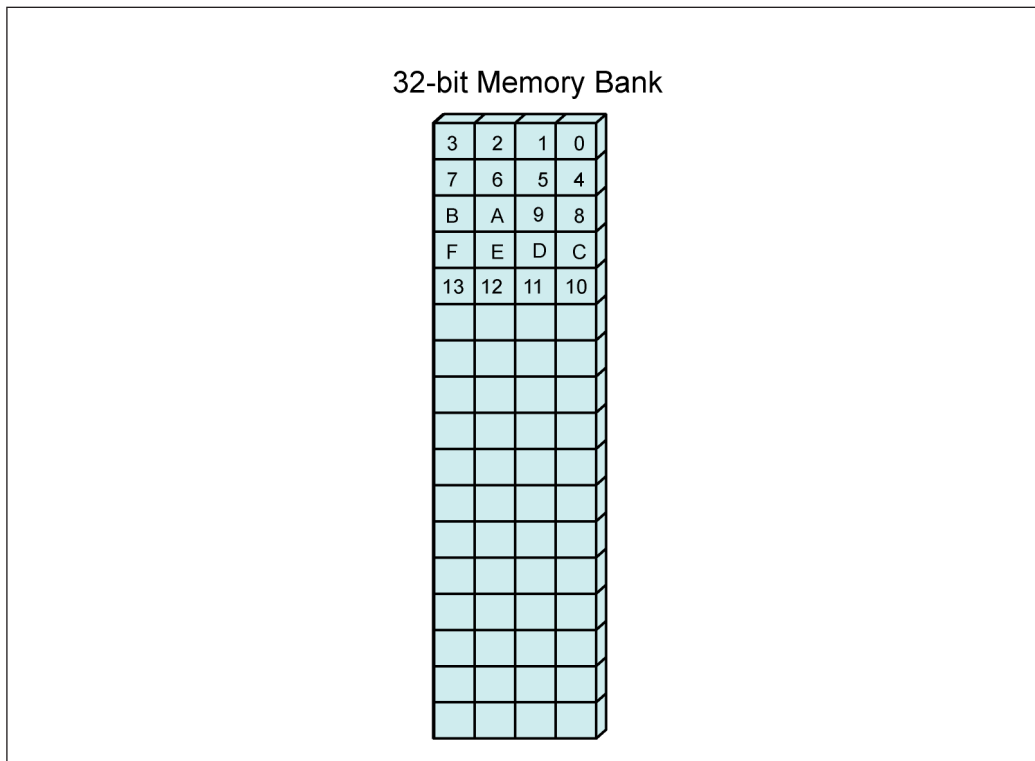
1. It initiates a memory read access to fetch the first (critical) dword. Wait states are inserted in the FSB transaction until the first dword is available for delivery.
2. The DRAM initiates the memory read by discharging all of the capacitors in that DRAM row to determine which bit cells contain 0s versus 1s.
3. The memory controller receives the first dword from memory and delivers it to the processor.

Chapter 1: Processor Hardware Feature Evolution

4. The row just addressed in the memory bank cannot be successfully accessed again until the DRAM has recharged the capacitors that were discharged within that row. This is referred to as the Precharge Delay. During this period of time, wait states are inserted in the FSB transaction.
5. After the Precharge Delay has elapsed, the memory controller can address the next dword in the requested line. Wait states are inserted in the FSB transaction until the second dword is available for delivery.

This sequence is repeated for the transfer of the four dwords that comprise the line read from memory. Having to access the same DRAM bank over and over results in poor performance.

Figure 1-7: Non-Interleaved Memory Architecture



Two-Way Interleaved Memory Architecture

Figure 1-8 on page 16 illustrates a system memory design wherein the DRAM is arranged in two banks, each of which is 32-bits (one dword) in width. The dwords are interleaved between the two banks:

- The dword that starts at address 00000000h resides in bank A.
- The dword that starts at address 00000004h resides in bank B.
- The dword that starts at address 00000008h resides in bank A.
- The dword that starts at address 0000000Ch resides in bank B.

Furthermore, the memory controller incorporates a 64-bit latch that can hold two dwords.

Refer to Table 1-3 on page 15. If the processor were to initiate a cache line fill (i.e., a memory read to fetch a 16-byte cache line), the address that is placed on the processor's address bus when the read is initiated is the start address of the critical dword. The memory controller takes the following actions to fulfill the request:

1. The memory controller simultaneously addresses both DRAM banks to fetch the critical dword and the adjacent dword in the opposite bank.
2. The two dwords are simultaneously read into the memory controller's 64-bit latch. The memory controller inserts wait states into the FSB transaction until the data becomes available.
3. Both banks then begin to charge back up.
4. The memory controller delivers the critical dword from the latch to the processor.
5. It then immediately delivers the adjacent dword from the latch to the processor. The wait states between the delivery of the first and second dwords have been eliminated.
6. By the time that the first two dwords have been delivered to the processor, both DRAM banks have completed charging back up and can be successfully accessed again.
7. The memory controller simultaneously addresses both DRAM banks to fetch the remaining two dwords from the two banks.
8. The two dwords are simultaneously read into the memory controller's 64-bit latch. The memory controller inserts wait states into the FSB transaction until the data becomes available.
9. Both banks then begin to charge back up.
10. The memory controller delivers the next dword from the latch to the processor.

Chapter 1: Processor Hardware Feature Evolution

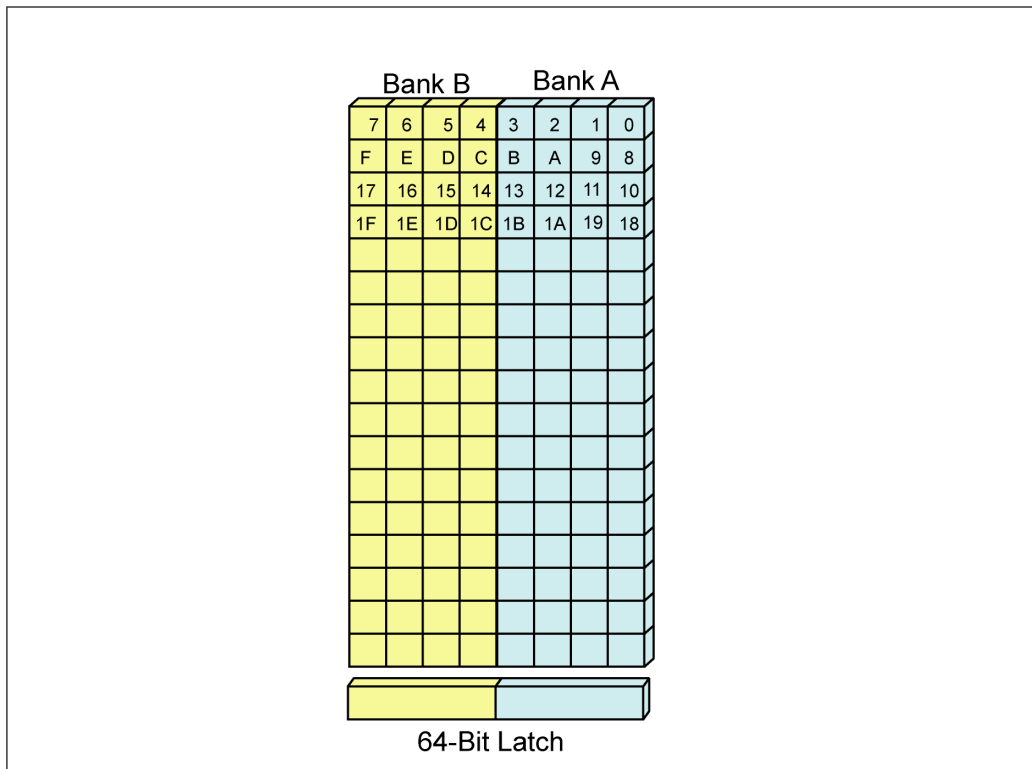
11. It then immediately delivers the adjacent dword from the latch to the processor. The wait states between the delivery of the last two dwords have been eliminated.
12. By the time that the last two dwords have been delivered to the processor, both DRAM banks have completed charging back up and can be successfully accessed again.

Table 1-3 on page 15 illustrates the order in which the processor expects to receive the four dwords that comprise a line. This transfer order is understood by the memory controller and is referred to as the toggle mode transfer order because, starting with the critical dword, the memory controller is essentially toggling back and forth between the two banks to fetch the requested line from memory.

Table 1-3: Cache Line Fill Toggle Mode Transfer Sequence

Start Address of Critical Dword	Dword Transfer Sequence			
00000000h	00000000h	00000004h	00000008h	0000000Ch
00000004h	00000004h	00000000h	0000000Ch	00000008h
00000008h	00000008h	0000000Ch	00000000h	00000004h
0000000Ch	0000000Ch	00000008h	00000004h	00000000h

Figure 1-8: Two-Way Interleaved Memory



Four-Way Interleaved Memory Architecture

Figure 1-9 on page 17 illustrates a system memory design wherein the DRAM is arranged in four banks, each of which is 32-bits (one dword) in width. The dwords that comprise a 16-byte line are striped across the four banks:

- The dword that starts at address 00000000h resides in bank A.
- The dword that starts at address 00000004h resides in bank B.
- The dword that starts at address 00000008h resides in bank C.
- The dword that starts at address 0000000Ch resides in bank D.

Furthermore, the memory controller incorporates a 128-bit latch that can hold an entire line.

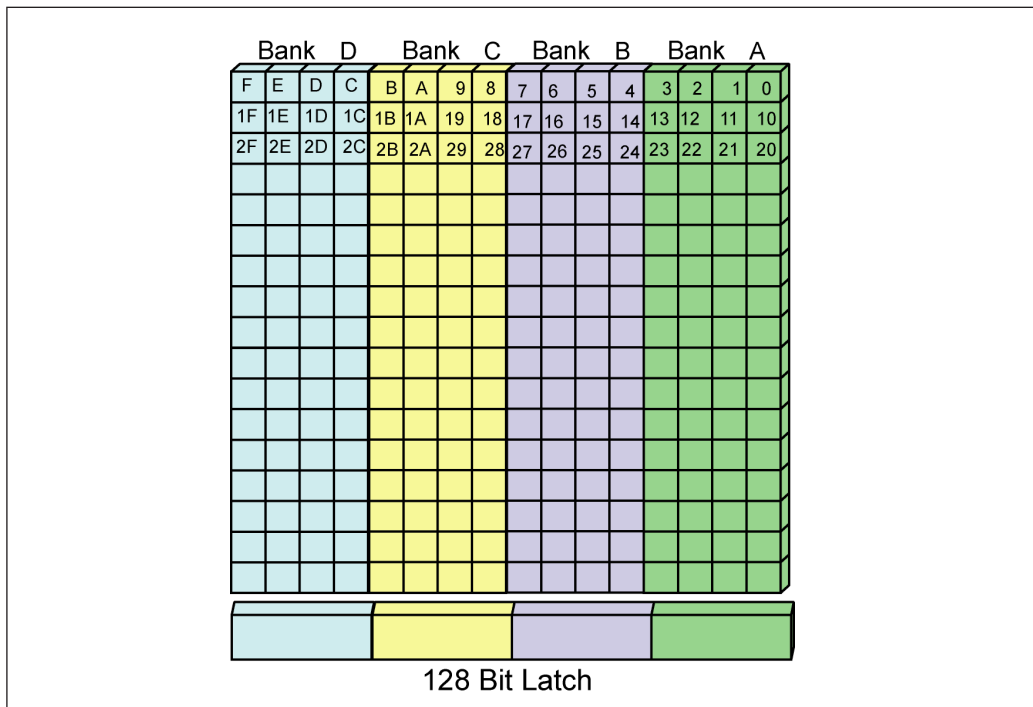
If the processor were to initiate a cache line fill (i.e., a memory read to fetch a 16-byte cache line), the address that is placed on the processor's address bus when

Chapter 1: Processor Hardware Feature Evolution

the read is initiated is the start address of the critical dword. The memory controller takes the following actions to fulfill the request:

1. The memory controller simultaneously addresses all four DRAM banks to fetch the critical dword as well as the remainder of the line in the other three DRAM banks.
2. The four dwords are simultaneously read into the memory controller's 128-bit latch. The memory controller inserts wait states into the FSB transaction until the data becomes available.
3. All four banks then begin to charge back up.
4. The memory controller delivers the critical dword from the latch to the processor.
5. It then immediately delivers the remaining three dwords from the latch in toggle mode transfer order. The wait states before the delivery of each of the remaining dwords have been eliminated.
6. By the time that the last dword has been delivered to the processor, all four DRAM banks have completed charging back up and can be successfully accessed again.

Figure 1-9: Four-Way Interleaved Memory



Problem: Separate FPU Is Slow

Problem Description

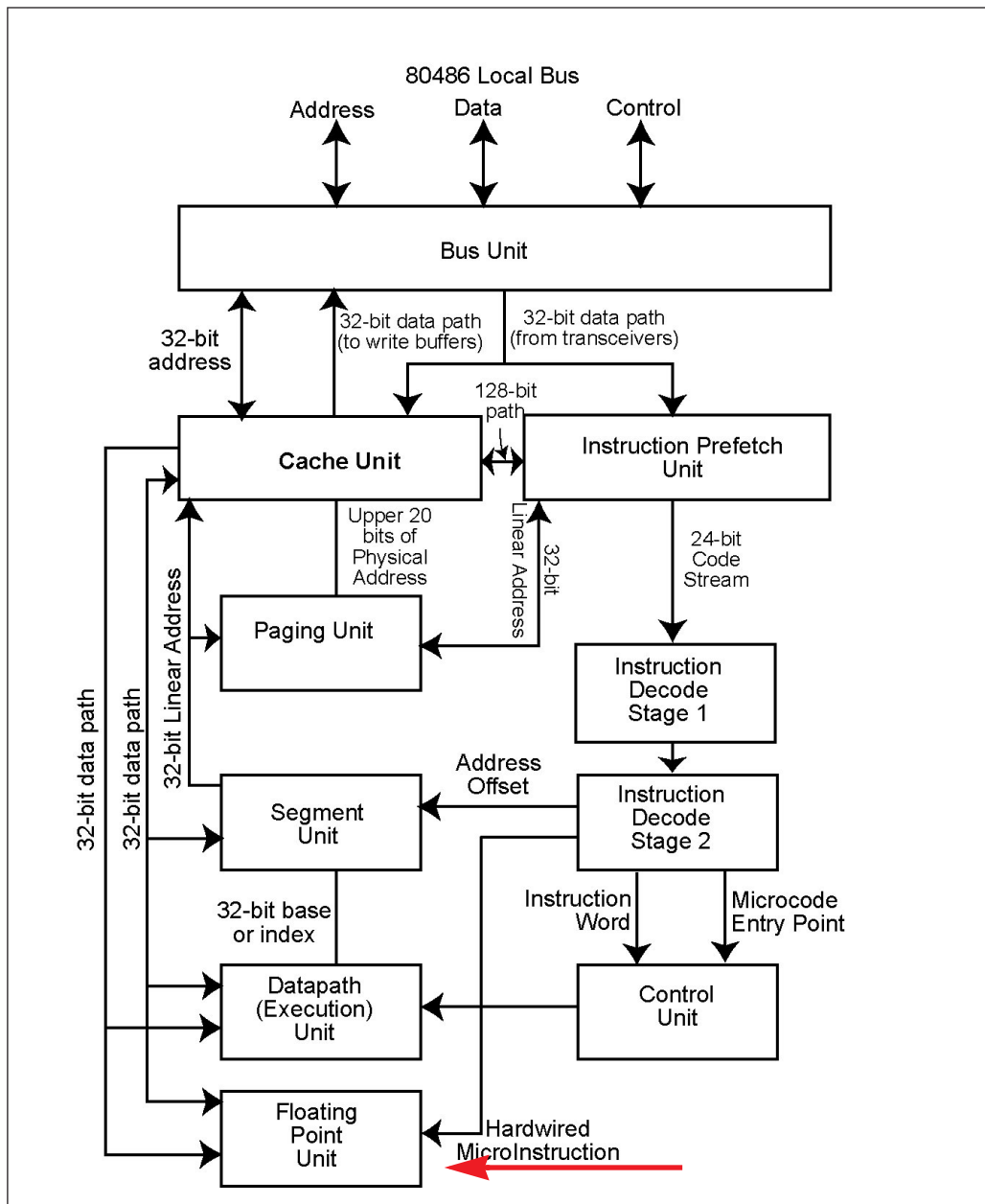
Prior to the 486 processor, the processors did not contain a Floating Point Unit (FPU). Rather, each processor had a companion x87 Numeric FP Coprocessor chip. If an instruction fetched from memory by the processor was a FP instruction, the processor would automatically perform an IO write transaction on its FSB to deliver the instruction to the x87 coprocessor for execution. This was quite inefficient and resulted in very poor FP performance.

Solution: Move FPU On-Chip

With the advent of the 486DX processor (see Figure 1-10 on page 19), the x87 Numeric Coprocessor was integrated onto the processor die (and is therefore no longer a coprocessor; rather, it is just another processor execution unit).

Chapter 1: Processor Hardware Feature Evolution

Figure 1-10: 486 Block Diagram



Problem: Memory Writes Are Slow

Problem Description

When the processor must perform a write to memory (i.e., a store operation), the processor execution unit stalls until the memory write data is delivered to the target memory location(s).

Solution: On-Chip Posted Write Buffer

Starting with the 486 processor, all IA32 processors implement an on-chip posted memory write buffer. When an instruction requires a write to memory, the address and data are latched into the on-chip Posted Memory Write (PMW) buffer. From the perspective of the Execution Unit, the memory write instruction completes immediately and the Execution Unit can begin execution of the next instruction. The posted writes are written to external memory at a later time (when the processor doesn't have a priority transaction to perform on the FSB, or when a synchronizing event is detected; a synchronizing event is one which causes the processor to update memory with all posted writes; a description of synchronizing events can be found in "Forcing a Buffer Drain" on page 1083 in the book).

Problem: Internal Cache Contention

Problem Description

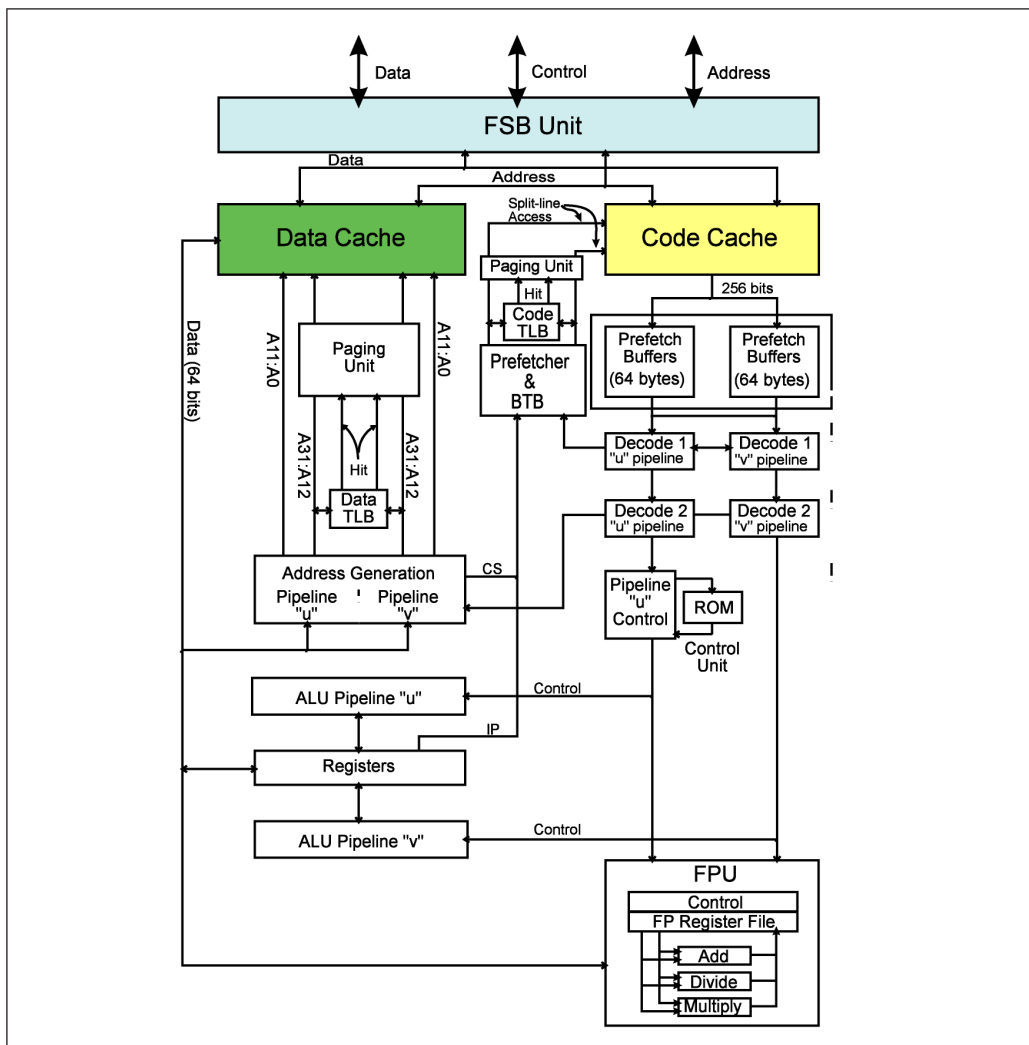
The 486 processor incorporated a unified code/data cache (i.e., a single cache that stores copies of both code and data lines). Although this yields performance improvements, the fact that the internal cache is used to store both data and code can lead to internal contention for access to the cache. Contention would occur when both the Instruction Prefetcher and the Execution Unit simultaneously require access to the cache. In that case, the Prefetcher is stalled while the Execution Unit's data access takes place.

Chapter 1: Processor Hardware Feature Evolution

Solution: Separate Data and Code Caches

Starting with the Pentium® processor (see Figure 1-11 on page 21), all IA32 processors incorporate separate (i.e., split) code and data caches, thereby solving the internal cache contention problem just described. The Execution Unit can submit a read or write request to the Data Cache at the same time that the code Prefetcher submits a read request to the Code Cache.

Figure 1-11: Pentium® Processor Block Diagram



Problem: Single Instruction Pipeline Becomes Choke Point

Problem Description

Up to and including the 486, all x86 processors employed a single instruction pipeline consisting of the following stages:

- Instruction fetch.
- Decode stages 1 and 2.
- Execution stage.
- Register write-back stage (wherein the results of an instruction's execution are written back to the appropriate processor registers).

Instructions proceeded through the pipeline one behind the other and were executed one at a time (i.e., the next instruction cannot move into the Execution Unit until the currently executing instruction completes).

Solution: Add Multiple Instruction Pipelines

The Pentium® processor implemented two, separate instruction pipelines (see Figure 1-11 on page 21), thereby allowing multiple instructions to complete execution per CPU clock cycle (referred to as superscalar operation):

- The Pentium® implemented two instruction pipelines, allowing it to complete the execution of two instructions per clock cycle.
- The Pentium® Pro, and the Pentium® 2, 3, and 4, as well as the Pentium® M processors are capable of completing the execution of three instructions per clock cycle.

Problem: Internal Cache Is Rather Small

Problem Description

Due to processor die size constraints, earlier IA32 processors were limited in the amount of on-chip cache that could be implemented. This meant that a limited amount of info (code and data) from system memory could be kept on chip for

Chapter 1: Processor Hardware Feature Evolution

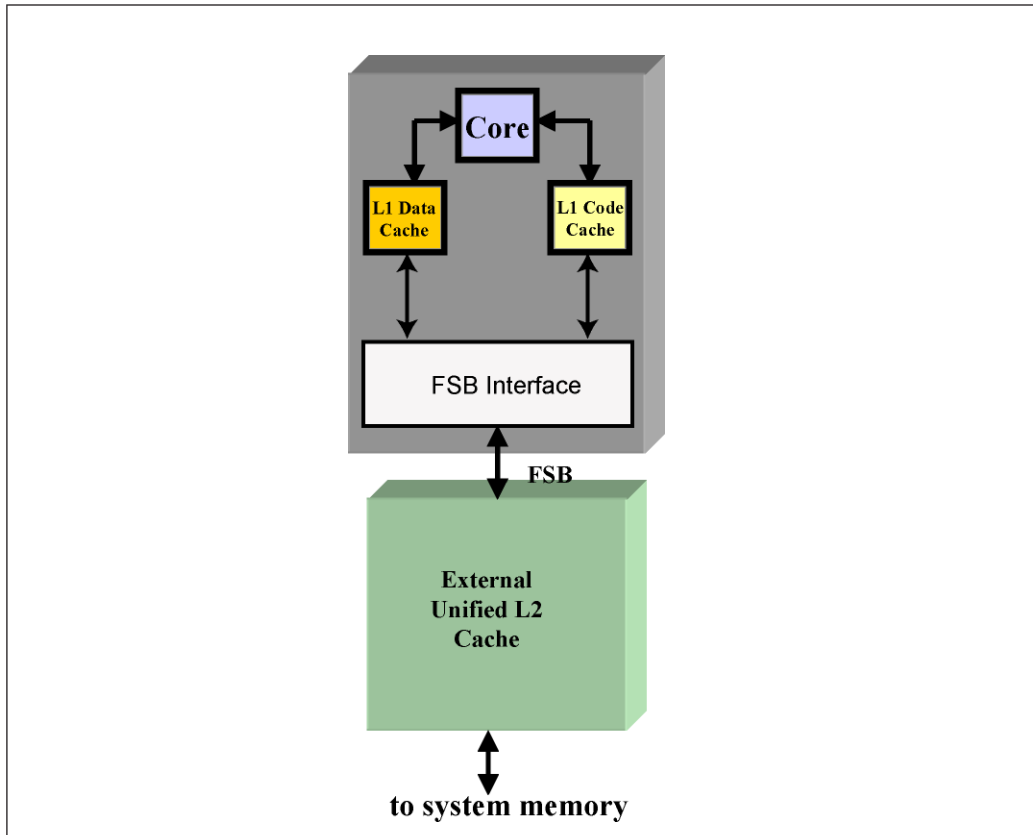
fast reference. Any code or data access that missed the on-chip cache necessitated the performance of a burst memory read transaction on the FSB to obtain the line containing the critical data item. Since the FSB runs at a substantially slower speed than the processor core, this resulted in poor performance.

Solution: Add External L2 Cache (first seen in 486 systems)

See Figure 1-12 on page 24. System board designers dealt with this problem by implementing an L2 Cache (and the on-chip cache became known as the L1 cache) external to the processor that is substantially larger than the on-die L1 cache. An L1 miss causes the initiation of a FSB burst memory read transaction. The L2 Cache uses the transaction's memory address to perform a lookup in the external L2 Cache:

- If the requested line is present in the L2 Cache, it is sourced to the processor at the full FSB speed (i.e., no wait states are inserted into the four Data Phases).
- If the lookup results in cache miss, the L2 Cache initiates a burst memory read transaction to fetch the desired line from system memory. As the four data items comprising the line are supplied to the L2 Cache by system memory, they are stored in the L2 Cache and are also immediately supplied to the processor for placement in the L1 cache.

Figure 1-12: Add an External L2 Cache



Problem: Prefetcher Can't Handle Branches

Problem Description

Prior to the advent of the Pentium® processor, the instruction Prefetcher always assumed that the Execution Unit would want the next instruction from the next sequential memory location. When the Execution Unit executes a branch instruction, however, the program flow is altered.

The Execution Unit would stall until the instruction from the branch instruction's target address was fetched and delivered to the instruction pipeline for

Chapter 1: Processor Hardware Feature Evolution

decode and execution. Processors prior to the Pentium® couldn't detect a jump to an instruction that was already resident in the pipeline or in the Prefetch Queue.

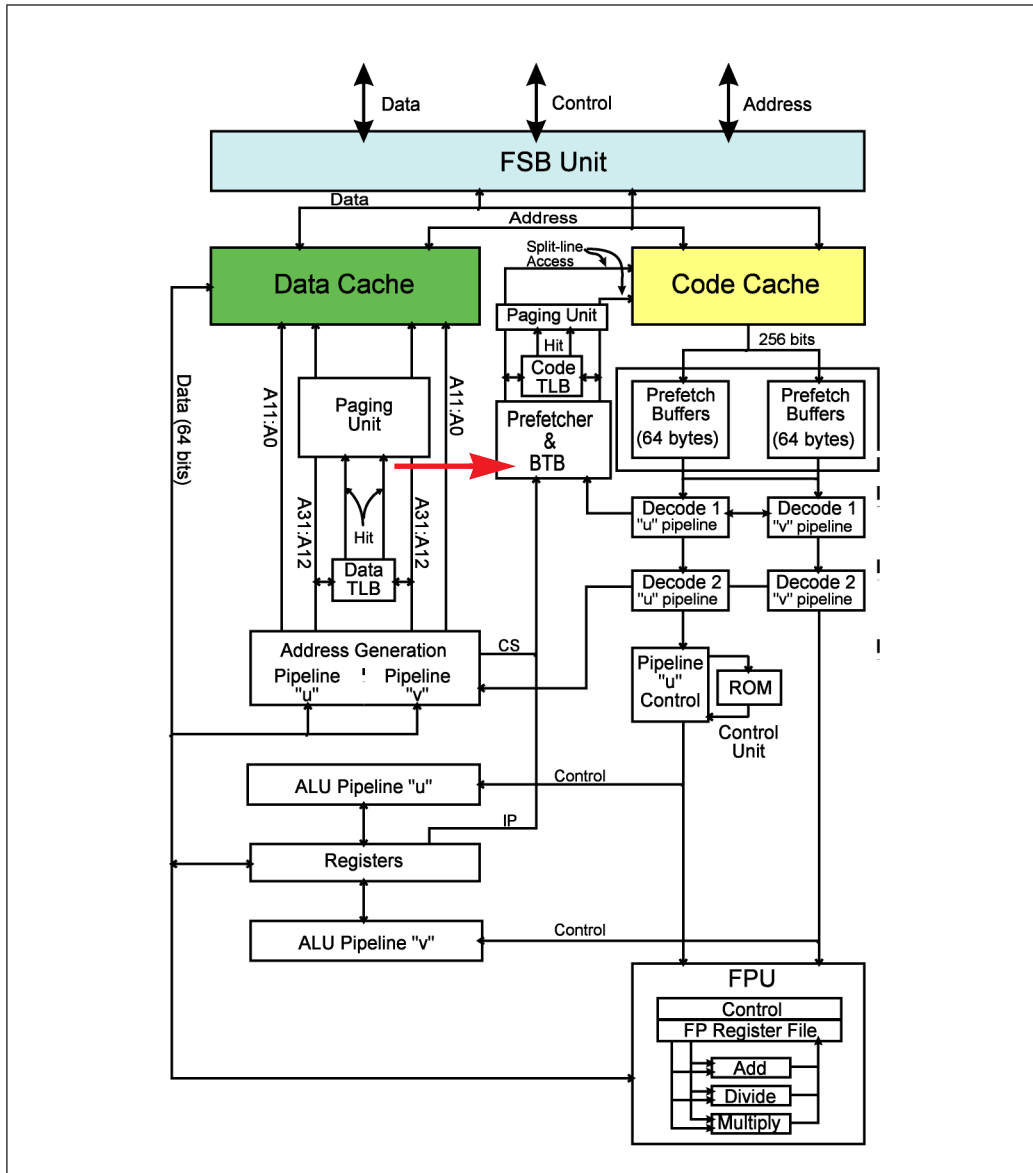
Solution: Add Branch Prediction Logic

Refer to Figure 1-13 on page 26. The Pentium® was the first IA32 processor to implement Branch Prediction Logic. It was implemented in the form of the Branch Target Buffer (BTB); a special-purpose cache that maintained an execution history for each branch instruction in the program being executed. When a branch is detected in the program stream being fetched from memory, the BTB uses the memory address from which the branch was fetched to perform a lookup in the BTB:

- A BTB miss indicates that the BTB does not contain any execution history for the branch. In this case, the Pentium® would predict that the branch will not be taken when it subsequently arrives at the Execution Unit and is executed. The Instruction Prefetcher therefore does not alter the path of its code fetching from memory (i.e., it continues to fetch instructions from memory sequentially).
- A BTB hit indicates that the BTB does have execution history for the branch. In this case, based on the prior history of that branch's execution, the Pentium® would either:
 - Predict that the branch will not be taken when it subsequently arrives at the Execution Unit and is executed. The Instruction Prefetcher therefore does not alter the path of its code fetching from memory (i.e., it continues to fetch instructions from memory sequentially).
 - Predict that the branch will be taken when it subsequently arrives at the Execution Unit and is executed. In this case, the instructions that currently reside in the Prefetch Queue are eliminated and the Prefetcher begins fetching instructions from memory starting at the branch's target address.

When the branch arrives at the Execution Unit and is executed, the BTB's history for that branch is updated to reflect the result of its latest execution. If the prediction was in error, this means that the instructions travelling through the pipeline after the branch are from the wrong path and they must be flushed (as must the instructions that currently reside in the Prefetch Queue). The Prefetcher is instructed to begin fetching instructions from the correct path. It should be noted that a mispredicted branch causes a bubble to be created in the instruction pipeline, resulting in a performance dip.

Figure 1-13: Pentium® Added Branch Prediction Logic



Chapter 1: Processor Hardware Feature Evolution

Problem: Single Processor Can Become a Choke Point

Problem Description

Uniprocessor systems have increased performance dramatically over the years. Some of the performance increase was obtained by increasing the processor's core clock speed and the efficiency of the processor core, as well as the efficiency of the FSB. Despite dramatic increases in performance, uniprocessor systems can become a major bottleneck to increased performance in many high-end environments. For example, a server running programs such as data warehousing and on-line transaction processing (OLTP) places heavy demands on the system processor.

When the fact that a system has only one processor becomes a bottleneck, the solution is to add more processors. This can be accomplished in any of the following ways:

- Traditional multiprocessing.
- Chip Multiprocessing (CMP).
- Hyper-Threading.

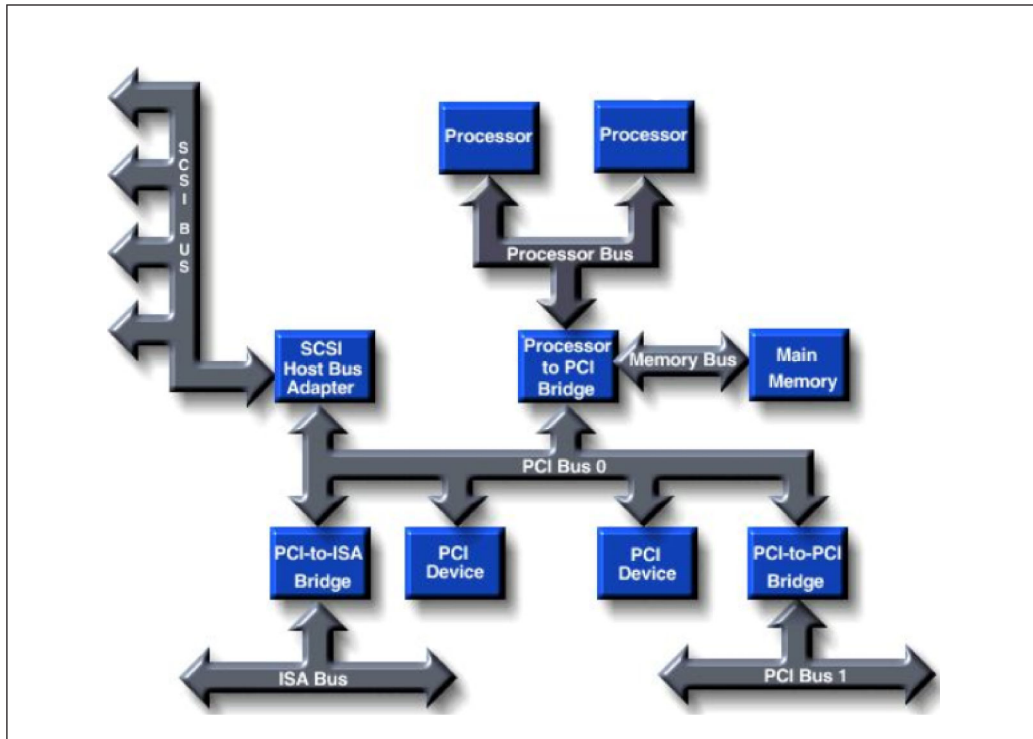
The next three sections introduce these three solutions.

Traditional Multiprocessing

Refer to Figure 1-14 on page 28. The system design may include multiple processors (MP) and an MP-aware OS. This is a VERY expensive solution in a number of ways:

- Each processor chip adds cost to the overall system.
- Each processor chip takes up system board real estate.
- Each processor chip consumes power.
- Each processor chip generates heat that must be exhausted from the system.

Figure 1-14: A System Incorporating Multiple Processor Chips



CMP

A processor chip can be designed so that the processor die implements two or more complete processor cores. This is referred to as CMP (Chip Multiprocessing) and it requires an MP-aware OS. While not as expensive as traditional multiprocessing, it is expensive. Example processors (as of this writing) that implement multiple cores are the IBM Power 4 processor and the Sun Microsystems UltraSparc IV processor.

Hyper-Threading

Hyper-Threading is also referred to as Simultaneous Multithreading and was first implemented in the Pentium® 4-based Xeon. It was migrated into the non-Xeon Pentium® 4 starting with the 3GHz processor (announced ~ 11/14/02).

Chapter 1: Processor Hardware Feature Evolution

Hyper-Threading permits the processor core to fetch and execute two, separate threads (code streams) simultaneously (so to speak). Two CS:EIP register pairs are implemented and the instruction prefetcher alternates fetching code for one program and then the other. The physical processor implements two, separate almost complete register sets so it can track the execution of two separate programs. There is only one set of execution units however. Detail on Hyper-Threading can be found in “Hyper-Threading” on page 965 in the book.

Problem: The x86 Instruction Set Is Complex

Problem Description

IA32 processors employ a variable-length instruction set. Depending on the instruction type and its operands, an instruction can be anywhere from one byte to 15 bytes in length. This complex instruction format requires complex decode and execution logic. This presents the processor designer with a formidable obstacle with regard to dispatching and executing more than one instruction simultaneously.

Solution: Go Primitive!

Starting with the P6 processor core in the Pentium® Pro, variable-length IA instructions are internally converted into fixed-length primitive instructions that are then executed by the processor core.

Problem: In-Order Execution Can Slow You Down

Problem Description

All IA32 processors prior to the Pentium® Pro were designed to always execute instructions in strict program order. This constraint limits the achievable throughput of the processor core.

Solution: Out-of-Order Execution

The Pentium® Pro's P6 core solved this problem as follows:

1. It fetches IA instructions from memory in strict program order.
2. It decodes (translates) them (still in strict program order) into one or more fixed-length primitive instructions known as micro-ops, or μ ops.
3. It places the μ ops into an instruction queue in strict program order.
4. It executes the μ ops in any order possible as the data operand(s) and execution unit required for each μ op become available.
5. It commits the results of each μ op's execution to the processor's register set in the order of the original program flow. The core of the Pentium® Pro, Pentium® II, Pentium® III, Pentium® 4, and Pentium® M processors can retire the results of three IA instructions per clock cycle to the processor's register set.

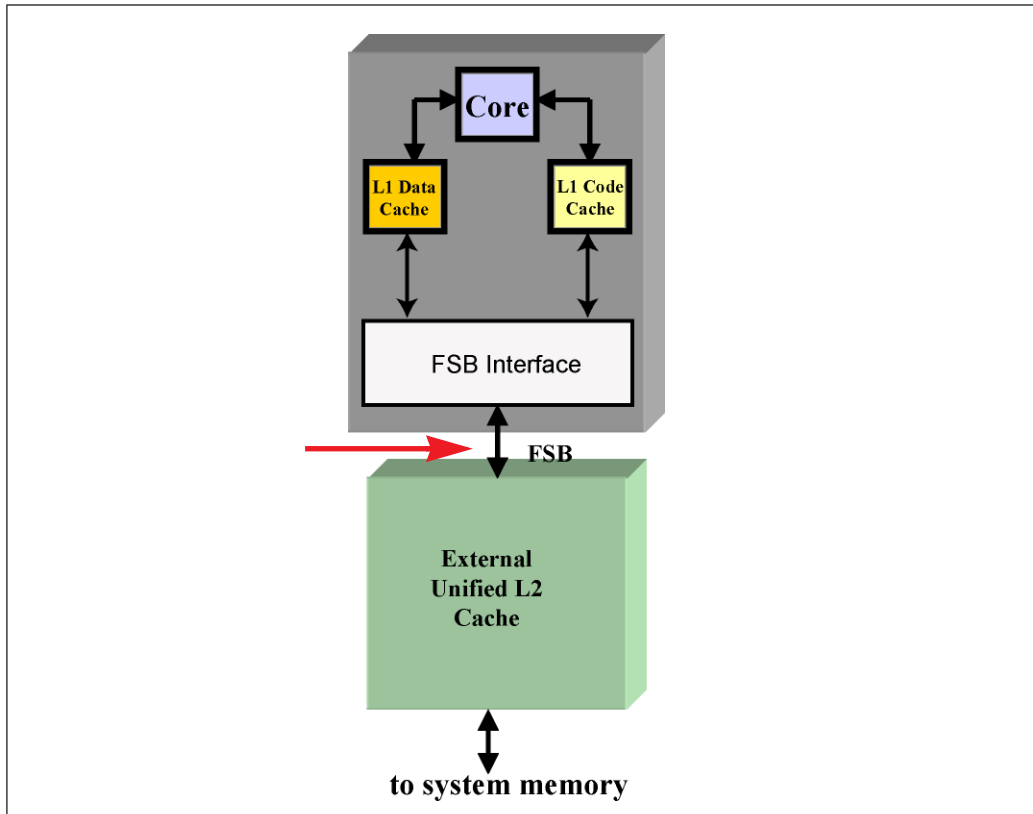
Problem: L2 Access Speed Bound by FSB Speed

Problem Description

Refer to Figure 1-15 on page 31. When the processor core experiences a miss on its L1 Code or L1 Data Cache, the processor is forced to perform a transaction on its FSB to obtain the target cache line. Since the FSB runs at a fraction of the processor core speed, the access to the external L2 Cache is quite slow and negatively affects the core's performance.

Chapter 1: Processor Hardware Feature Evolution

Figure 1-15: Add an External L2 Cache



Solution: Move the L2 Closer to the Processor Core

Pentium® Pro

Refer to Figure 1-16 on page 33. The Pentium® Pro processor employed a unique, dual-cavity design wherein the L2 Cache die occupied one cavity and the processor core die the other cavity. Whenever the core experienced a miss on the L1 Data or L1 Code Cache, the access request is forwarded to the L2 Cache over the a dedicated Back Side Bus (BSB) that ran at the full processor core speed. This had two major advantages:

- The FSB remains available for the processor (or another processor or device) to use.
- The speed of the BSB (relative to the FSB speed) yielded substantially better

The Unabridged Pentium® 4

transfer rates than the arrangement shown in Figure 1-15 on page 31.

The fact that the processor now has two separate bus interfaces is sometimes referred to as the Dual Independent Bus Architecture (DIBA).

Pentium® II

The Pentium® II processor package shifted to the cartridge format (see Figure 1-17 on page 33). While it also follows the DIBA paradigm, the L2 Cache is implemented using discrete chips mounted on the cartridge circuit board (see Figure 1-18 on page 34 and Figure 1-19 on page 34). Rather than running at the full processor core speed, the BSB runs at 50% of the core speed. While still substantially faster than the processor's FSB, it was slower than the core speed. When the core experiences an L1 Code or L1 Data Cache miss, the access request is forwarded to the L2 Cache over the slow BSB, yielding poor performance.

Solution: Move the L2 Onto the Processor Die

With the advent of the Celeron Mendocino processor, the L2 Cache was moved onto the processor die. All subsequent IA32 processors integrate this feature. The resulting, extremely short BSB trace lengths and tightly controlled electrical environment on the processor die permit the BSB to run at the full processor core speed.

Chapter 1: Processor Hardware Feature Evolution

Figure 1-16: Pentium® Pro Packaging

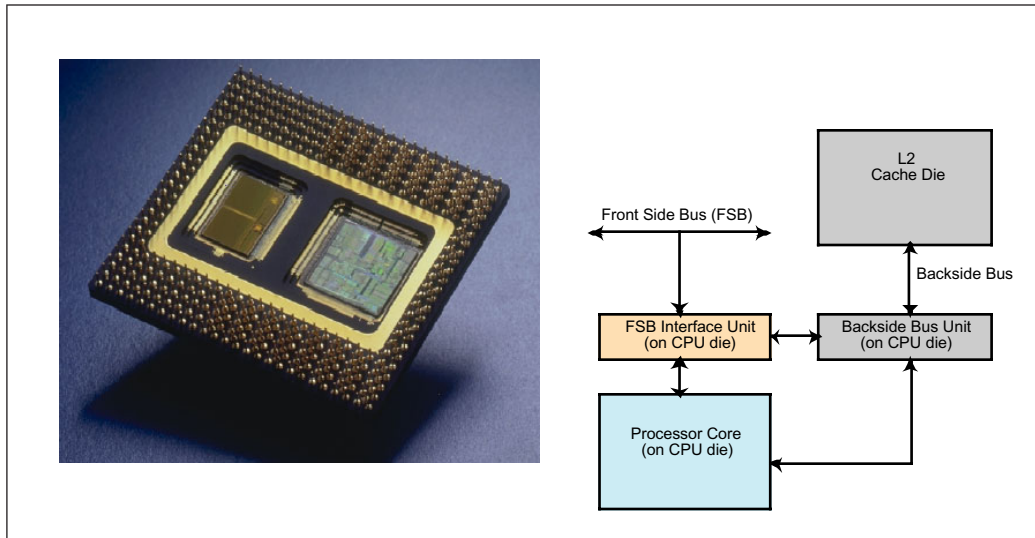


Figure 1-17: Pentium® II Cartridge



The Unabridged Pentium® 4

Figure 1-18: Pentium® II Cartridge: View 1

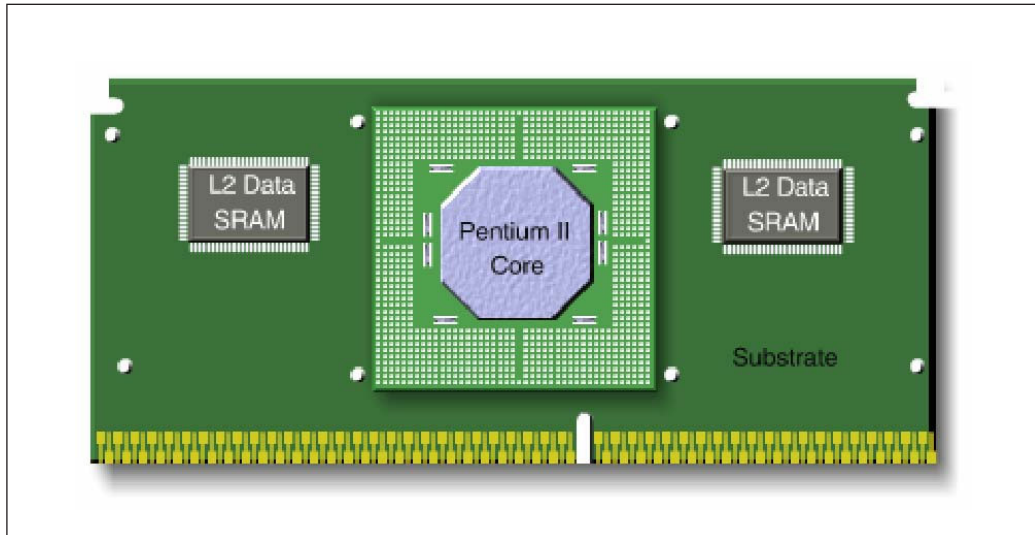
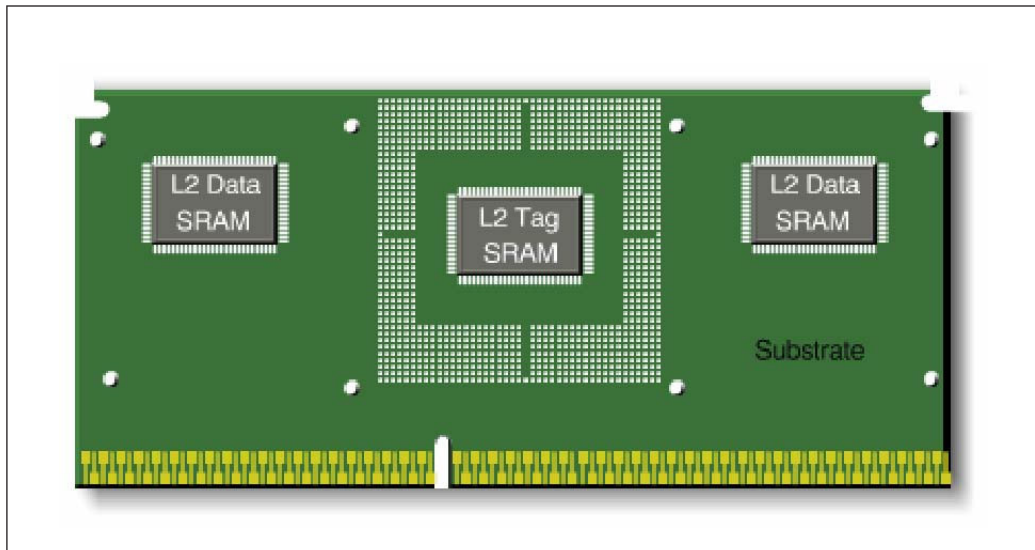


Figure 1-19: Pentium® II Cartridge: View 2



Chapter 1: Processor Hardware Feature Evolution

Problem: Math Units Ill-Suited for Array Operations

Problem Description

Prior to the advent of the Pentium® processor, Intel® processors were not able to perform parallel (simultaneous) operations on packed data (e.g., video image data). Rather, the application programmer was responsible for:

- Reading packed data from memory.
- Masking the packed data to isolate the next numeric data item to be processed.
- Performing the desired operation on the data item.
- Melding the updated data item back into the packed data.
- Repeating these operations on each data item in the packed data (taking any overflow from the previous operation into account).
- Storing the resultant, updated packed data back into memory.

All of this complicated the programmer's life and consumed an exorbitant number of processor clock cycles.

Solutions

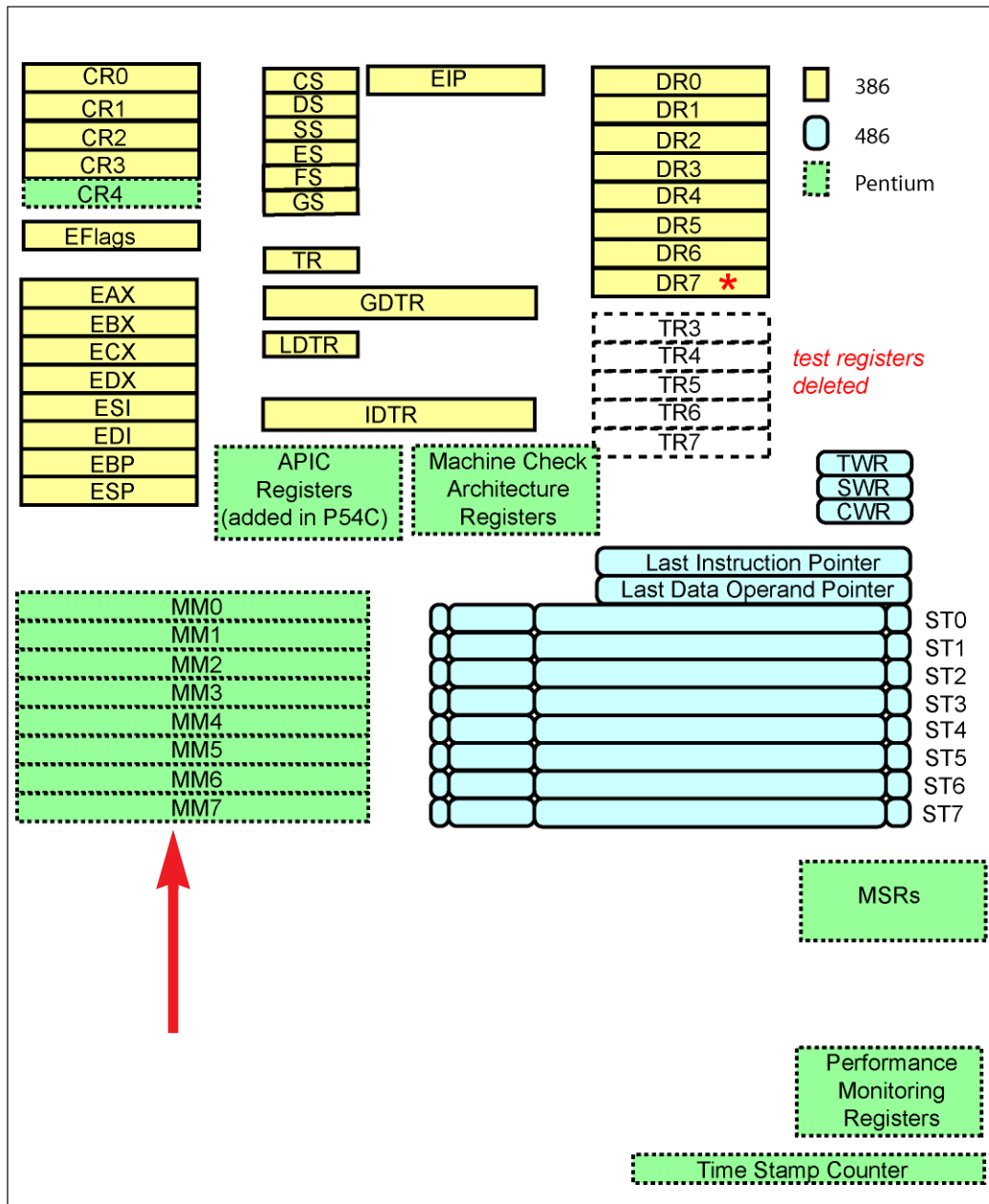
MMX Instruction Set and Registers

Refer to Figure 1-20 on page 36. With the advent of the P55C version of the Pentium® processor, Intel® included the MMX instruction set, capable of SIMD (Single Instruction operating on Multiple Data items) operations wherein a single instruction simultaneously operates on multiple packed data items. All subsequent Intel® processors (with the exception of the Pentium® Pro) include the MMX instruction set and a set of 64-bit MMX data registers.

A more detailed discussion of MMX can be found in "MMX Capability" on page 519 in the book.

The Unabridged Pentium® 4

Figure 1-20: Pentium® Register Set Includes a Set of 64-bit MMX Data Registers



Chapter 1: Processor Hardware Feature Evolution

SSE Instructions and Registers

Refer to Figure 1-21 on page 38. The Pentium® III processor's Katmai core expanded the MMX instruction set to include 70 new instructions (referred at the time as Katmai New Instructions, or KNI) divided into two categories:

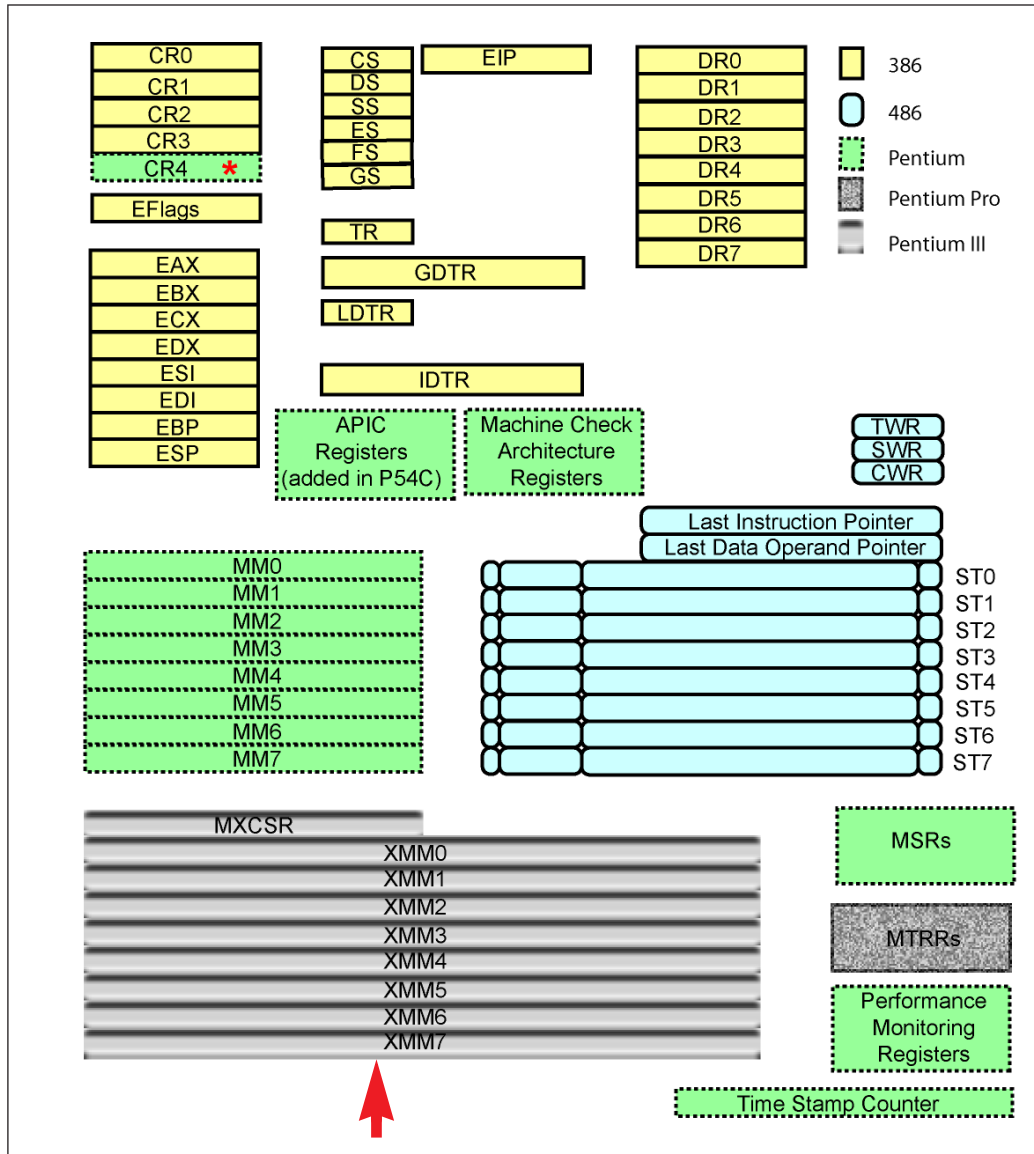
- New Media instructions to enhance applications such as speech recognition, video decoding, and motion search.
- SIMD single-precision (SP) FP capability.

Later, these were referred to as the ISSE (Internet Streaming SIMD Extensions), or just SSE for short. In addition to the SSE instructions, the Pentium® III also added an SSE Control/Status register (MXCSR) and eight new 128-bit FP data registers, each capable of holding four, 32-bit IEEE SP FP data values. However, the internal data paths used to access these registers were only 64 bits wide, resulting in a register access time of two processor clock cycles.

A more detailed description of the SSE instructions can be found in "The Streaming SIMD Extensions (SSE)" on page 756.

The Unabridged Pentium® 4

Figure 1-21: Pentium® III Register Set



SSE2

The Pentium® 4 processor core added the SSE2 instructions (no new registers were added), further enhancing both integer and FP SIMD operations. Among

Chapter 1: Processor Hardware Feature Evolution

other capabilities, SSE2 added the ability to perform SIMD operations on packed 64-bit double-precision (DP) FP data.

A more detailed description of the SSE2 instructions can be found in “The SSE2 Instruction Set” on page 1332 in the book.

SSE3

The 90nm Pentium® 4 processor (code named Prescott) included an additional 13 instructions referred to as the SSE3 instruction set (code named Prescott New Instructions, or PNI). A more detailed description of the SSE3 instructions can be found in “The SSE3 Instruction Set” on page 1337 in the book.

Problem: Execution Speed of Simple Math Operations

Problem Description

In earlier IA32 processors, the fastest the processor core could execute simple math operations such as Add and Subtract was a single processor clock cycle.

Solution: Double-Clock Simple ALUs

The Pentium® 4 processor has two ALUs (Arithmetic Logic Units) each of which is capable of executing a simple operation in one half of a processor clock cycle. These are sometimes referred to as double-pumped ALUs.

Problem: L1 Code Cache Is Inefficient

Problem Description

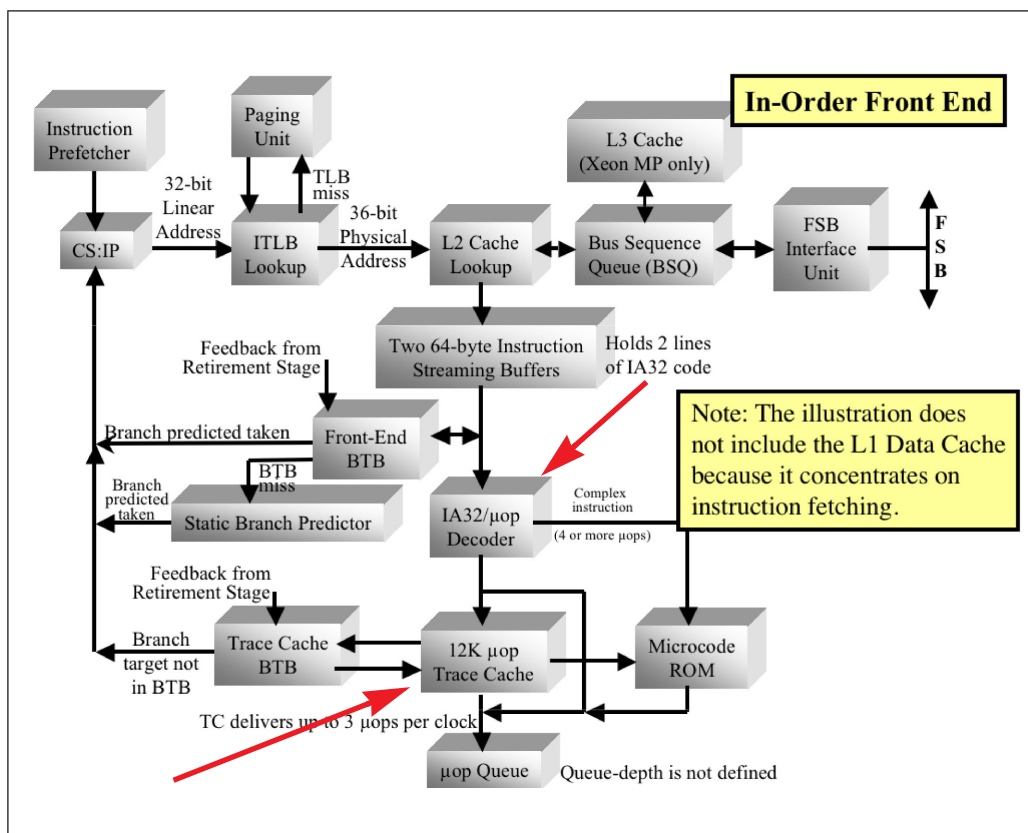
Prior to the advent of the Pentium® 4 processor, the processor’s L1 Code Cache maintained copies of variable length IA instructions rather than the μ ops that they are decoded into. For this reason, each time that a line of code was fetched from the L1 Code Cache it would have to be decoded into μ ops before being executed.

Solution: Execution Trace Cache

Refer to Figure 1-22 on page 40. The Pentium® 4 processor replaced the L1 Code Cache with a Trace Cache capable of holding up to 12K μ ops. When IA instructions are first fetched from memory and decoded into μ ops, those μ ops are cached in the Trace Cache. When the processor core requires an instruction, it first performs a lookup in the Trace Cache. In the event of a Trace Cache hit, the μ op(s) are fed directing to the core, bypassing the IA32 instruction decode stage.

In the event of a Trace Cache miss, a lookup is performed in the L2 Cache (which contains IA instructions).

Figure 1-22: The Pentium® 4 Added the Trace Cache



Chapter 1: Processor Hardware Feature Evolution

Problem: Some Applications Deal With Massive Databases

Problem Description

When executing a data warehousing application (e.g., an airline reservations system), the processor must have rapid access to extremely large amounts of data. The processor's on-die L1 and L2 Caches are relatively small in size because they are constrained by the amount of available die space. A data fetch that misses the L1 and L2 Caches results in a cache line read transaction performed on the processor's FSB. Since the FSB runs at a substantially slower frequency than the processor core, and the access time of system memory may be relatively slow, this results in performance degradation.

Solution: Add External L3 Cache

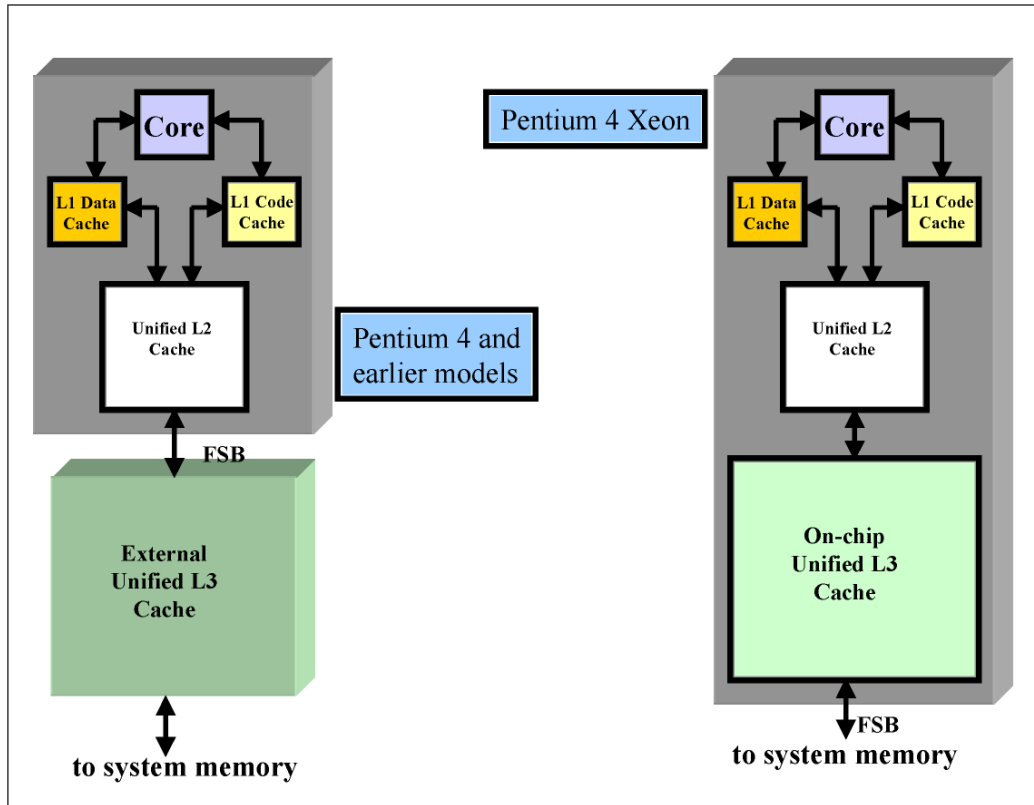
Refer to Figure 1-23 on page 42. The left side of the illustration shows a processor without an on-die L3 cache. An external L3 cache has been added to the system design, but it is accessed at the speed of the processor's FSB (which is only a fraction of the processor core speed).

Solution: Move L3 Cache On-Chip

Refer to the right side of Figure 1-23 on page 42. The Pentium® 4 MP (MultiProcessing version) was the first IA32 processor to bring the L3 cache on board the processor die. This permits accesses to the L3 cache at the full processor core speed. As of this writing (October 2003), Intel® has begun to migrate the on-die L3 cache into a limited number of other processor models (e.g., The Pentium® 4 Extreme). It should be noted that although this provides a considerably higher level of performance, it also comes at a considerably higher cost.

The Unabridged Pentium® 4

Figure 1-23: Off-Chip and On-Chip L3 Cache



Problem: FSB Not Optimized for Multiprocessing

Problem Description

The FSB design on the earlier IA32 processors was not designed to efficiently handle multiple processors residing on the same FSB. In those earlier processor designs, the FSB could only be used by one initiator at a time. Another device could not initiate a FSB transaction until the current initiator's transaction completed (and, if a slow device is being accessed, the transaction currently in progress could take a while to complete).

Chapter 1: Processor Hardware Feature Evolution

While this type of FSB design is acceptable in a single-processor system, it is terrible in a system that incorporates multiple processors on the FSB bus.

The Pentium® processor's FSB had a slightly improved FSB in that a device could initiate a transaction before the transaction currently in progress completed. However, a third transaction could not be initiated until the first transaction completed. In other words, no more than two transactions could be pipelined onto the Pentium® processor's FSB.

Solution: FSB Redesign

Refer to Figure 1-24 on page 44. The Pentium® Pro FSB was specifically designed to efficiently support multiple processors residing on the same FSB. The following major changes were made:

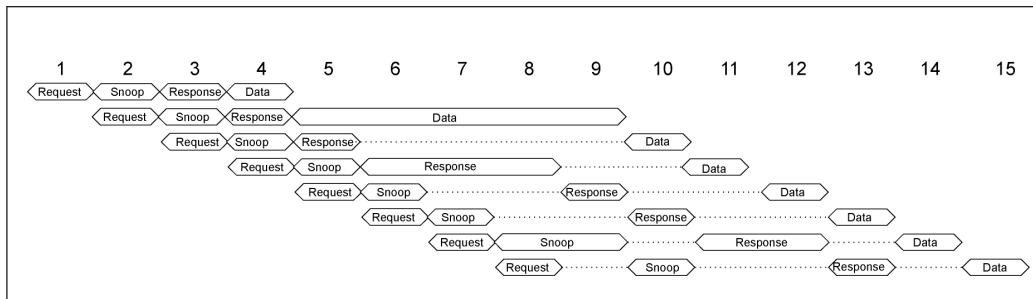
- Up to eight transactions can simultaneously be in progress at various stages of completion.
- If the target of a transaction cannot deal with a new transaction right now (e.g., due to a temporary logic busy condition), it issues a retry to the initiator.
- If the target of a read or write transaction realizes that it will take a fairly long time to complete the data transfer (i.e., provide read data or accept write data):
 - It can issue a Deferred response to the initiator of the transaction. This indicates that the target has memorized the transaction request (including the identity of the initiator and the transaction ID).
 - The request initiator moves the transaction into its Deferred Transaction Queue.
 - When the target has the requested read data available (on a read) or has completed the write, it arbitrates for ownership of the FSB and initiates a Deferred Reply transaction. It addresses the processor that originated the request and provides the transaction ID:
 - On a read, the requested read data and completion status is delivered, completing the transaction.
 - On a write, the completion status is delivered, completing the transaction.

These mechanisms prevent any initiator from tying up the FSB for extended periods of time. The Pentium® 2, 3, 4 and M all use variations on the Pentium® Pro processor's FSB. It should be noted that the Pentium® 4 processor FSB design supports the pipelining of up to 12 transactions.

The Unabridged Pentium® 4

A detailed description of the variations on the FSB can be found in the sections on the various processors (starting with the Pentium® Pro).

Figure 1-24: P6 FSB Transaction Pipelining



Problem: The IA Register Set Is Small

Problem Description

In pre-Pentium® Pro IA32 processors, the subset of the IA register set that could be used to hold data items was extremely small. As a result, the programmer was frequently forced to write back the contents of one or more of the processor's registers to memory when additional data operands must be read from memory to be operated on.

Later, when the programmer requires access to the original set of data operands, they must again be read from memory (perhaps after first storing the current contents of the registers).

This juggling of data between the register set and memory takes time and exacts a penalty (perhaps severe) on the performance of the program. In addition, a considerable of FSB bandwidth is consumed by the resultant memory transactions. Figure 1-25 on page 46 illustrates the portion of the IA general register set that could be used to hold data items.

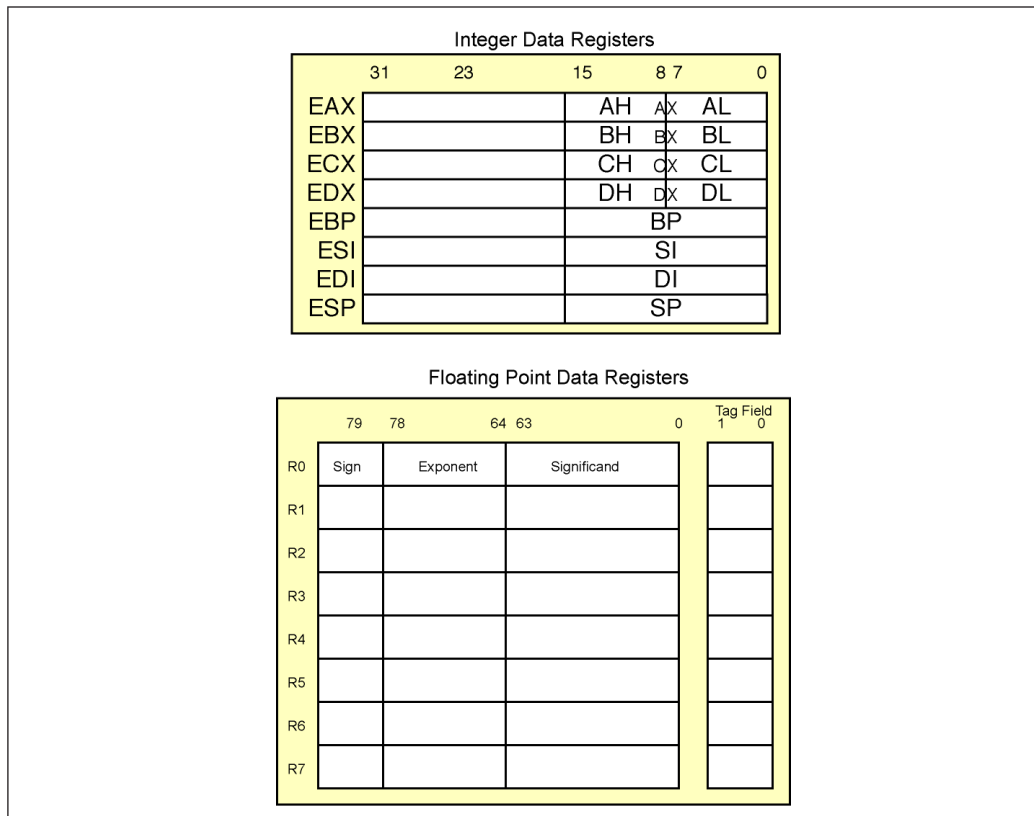
Solution: Instruction Register References Are Mapped To a Large Physical Register Set

Rather than the extremely limited data register set pictured in Figure 1-25 on page 46:

- The Pentium® Pro and Pentium® II processors implemented 40 integer data registers and the eight x87 FP data registers.
- The Pentium® III processor implemented 40 integer data registers, eight XMM data registers, and the eight FP data registers.
- The Pentium® 4 implements 128 integer data registers, 128 FP data registers, and eight x87 FP data registers.

A detailed description of the register remapping can be found in the sections on the Pentium® Pro and the Pentium® 4 processors.

Figure 1-25: IA Data Registers



Problem: Cache Miss Causes Cessation of Lookups

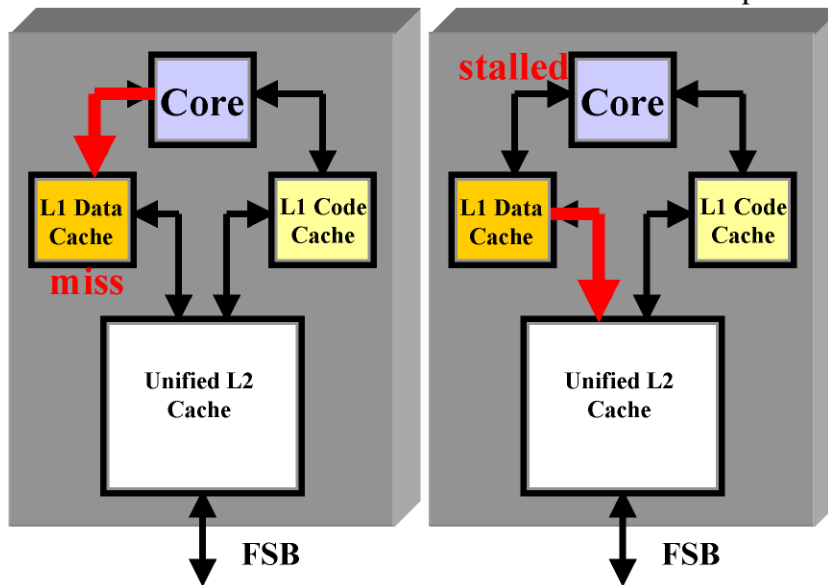
Problem Description

Refer to Figure 1-26 on page 47. In IA32 processors prior to the Pentium® Pro, a miss on the processor's cache resulted in a processor stall—no further memory requests could be submitted to the cache until the completion of the cache line fill from external memory.

Chapter 1: Processor Hardware Feature Evolution

Figure 1-26: Blocking Cache

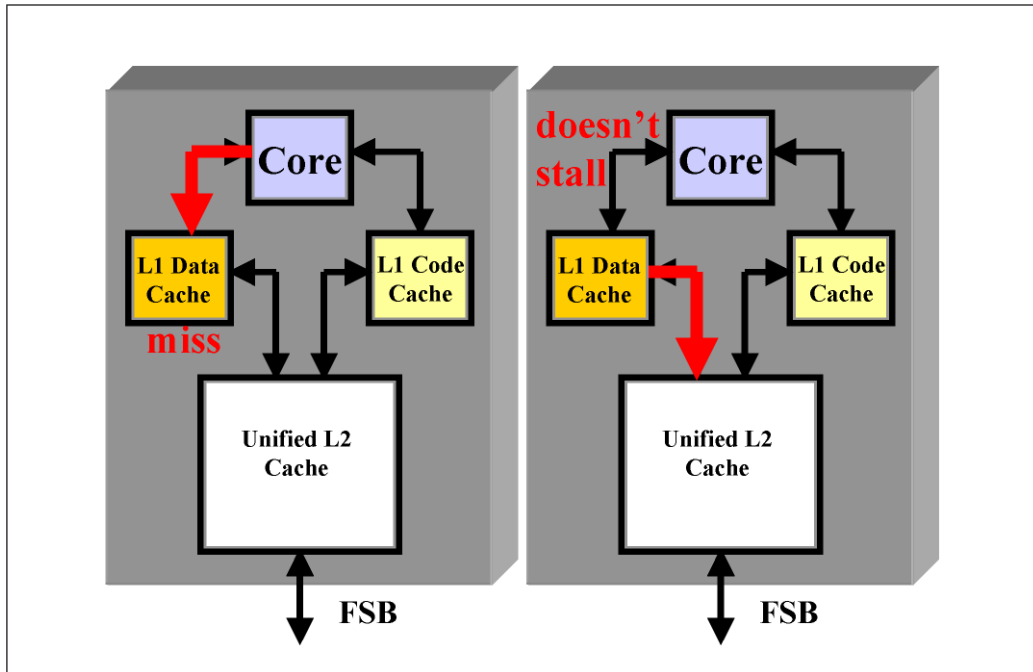
1. Load instruction causes data cache lookup resulting in a miss.
2. Request forwarded to L2 cache for lookup. Core cannot execute any other loads until stalled load completes.



Solution: Non-Blocking Caches

Refer to Figure 1-27 on page 48. When a miss occurs on the data cache in any IA32 processor starting with the Pentium® Pro, however, the request is forwarded to the processor's L2 Cache and the L1 Data Cache is immediately available to service memory access requests (i.e., it does not block them) that might be generated by the execution of subsequent load or store instructions in the program.

Figure 1-27: Non-Blocking Cache



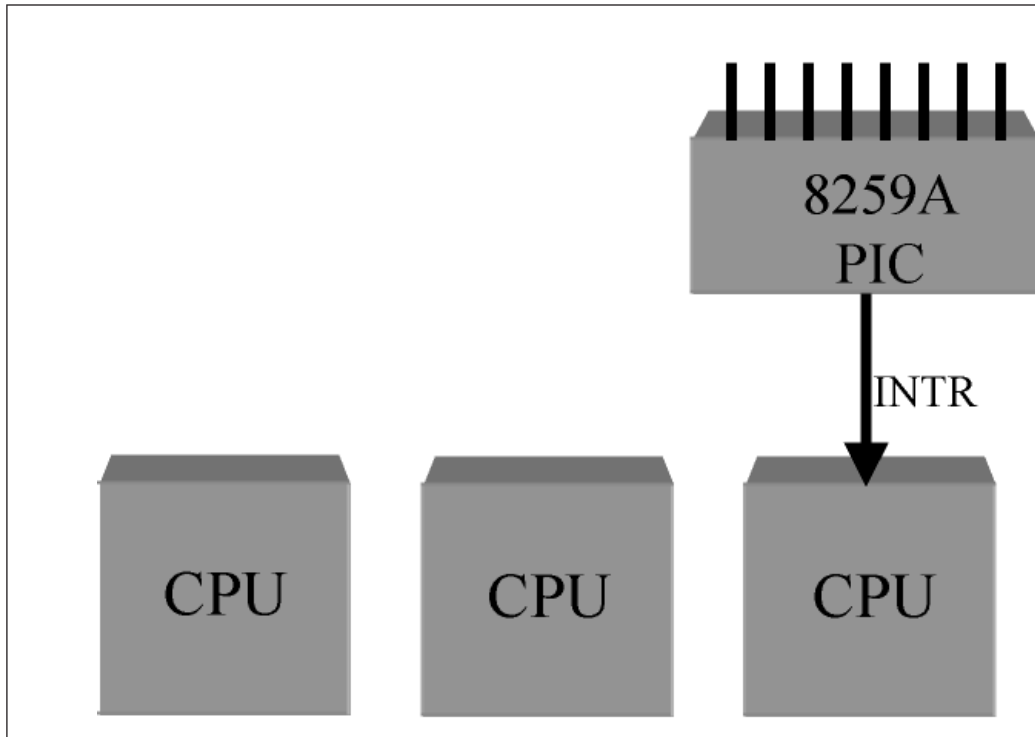
Problem: Interrupt Delivery in Multiprocessor Environment

Problem Description

Refer to Figure 1-28 on page 49. The earliest PCs implementing multiple processors relied on the Intel® 8259 Programmable Interrupt Controller (PIC) to deliver interrupt requests generated by hardware devices. The 8259 was designed for single-processor implementations and has no inherent ability to direct interrupts to different processors. This means that rather than providing a symmetric multiprocessing environment wherein any processor can execute any task, this type of system design provided an asymmetric multiprocessing environment because one processor was tasked with servicing all interrupt requests generated by hardware devices.

Chapter 1: Processor Hardware Feature Evolution

Figure 1-28: Old-Style Interrupt Delivery



Solution: APIC

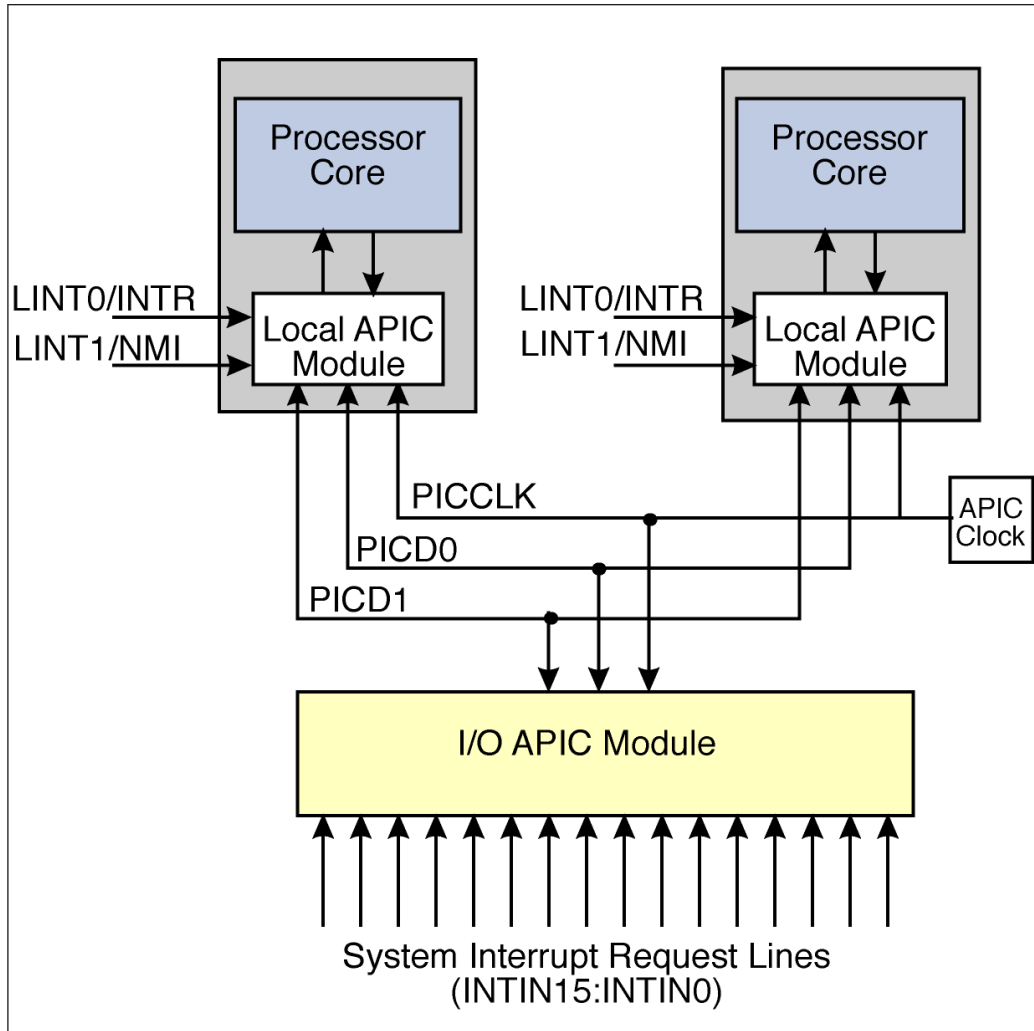
Refer to Figure 1-29 on page 50. The Advanced Programmable Interrupt Controller (APIC) was first integrated in the P54C version of the Pentium® processor and provides interrupt support for both symmetrical and asymmetrical multiple processing environments via its numerous operational modes.

The system board design includes an IO APIC module that receives interrupt requests from the various hardware devices in the system. Each of the processors includes an integrated Local APIC module. Via a three-wire APIC bus (consisting of two data lines and a clock signal), the IO APIC module is connected in a daisy chain manner to the Local APICs in each of the processors. The IO APIC is programmed by the OS, telling it to which processor each of the hardware interrupt requests are to be delivered. When an interrupt request is received from a hardware device, the IO APIC generates an interrupt message and directs it to a processor.

The Unabridged Pentium® 4

A detailed discussion of the APIC can be found in "The Local and IO APICs" on page 1497 in the book.

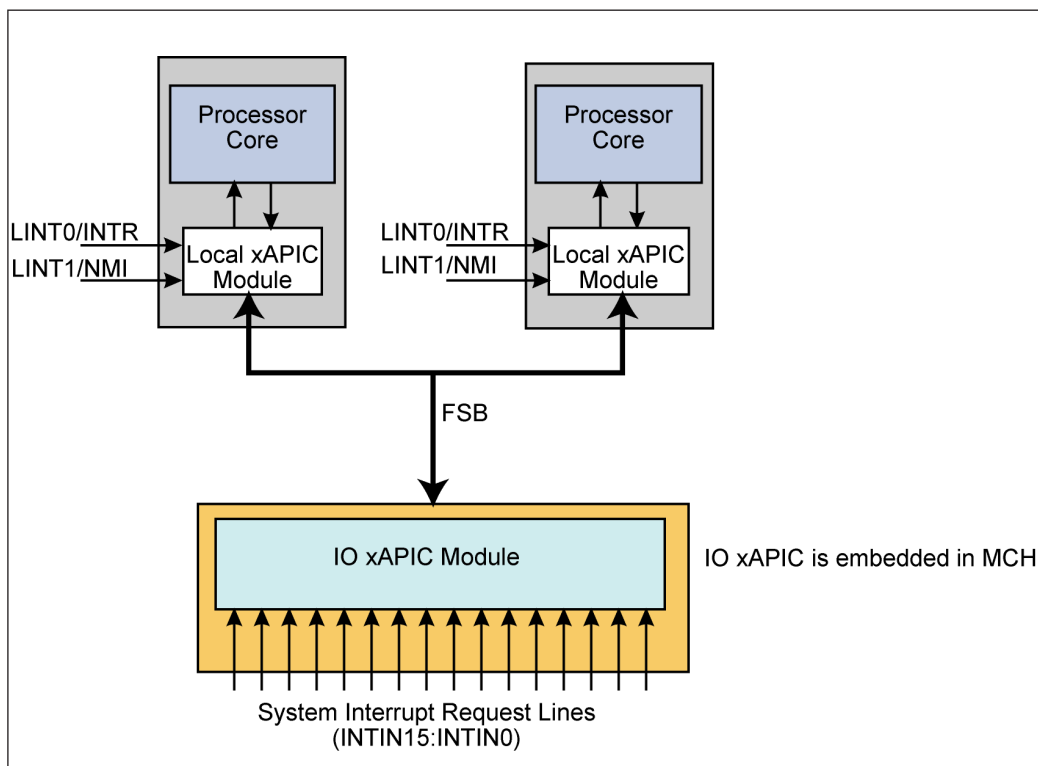
Figure 1-29: Distributed Interrupt Delivery



APIC Evolves in the Pentium® 4

Refer to Figure 1-30 on page 51. With the advent of the Pentium® 4 processor, the 3-wire APIC bus was eliminated and interrupt messages are transferred between the IO APIC and the processor's local APIC (or between the local APICs in different processors) using a new transaction type on the FSB, the interrupt transaction.

Figure 1-30: The Pentium® 4 Improved the Interrupt Delivery Mechanism



Problem: No Data Prefetching

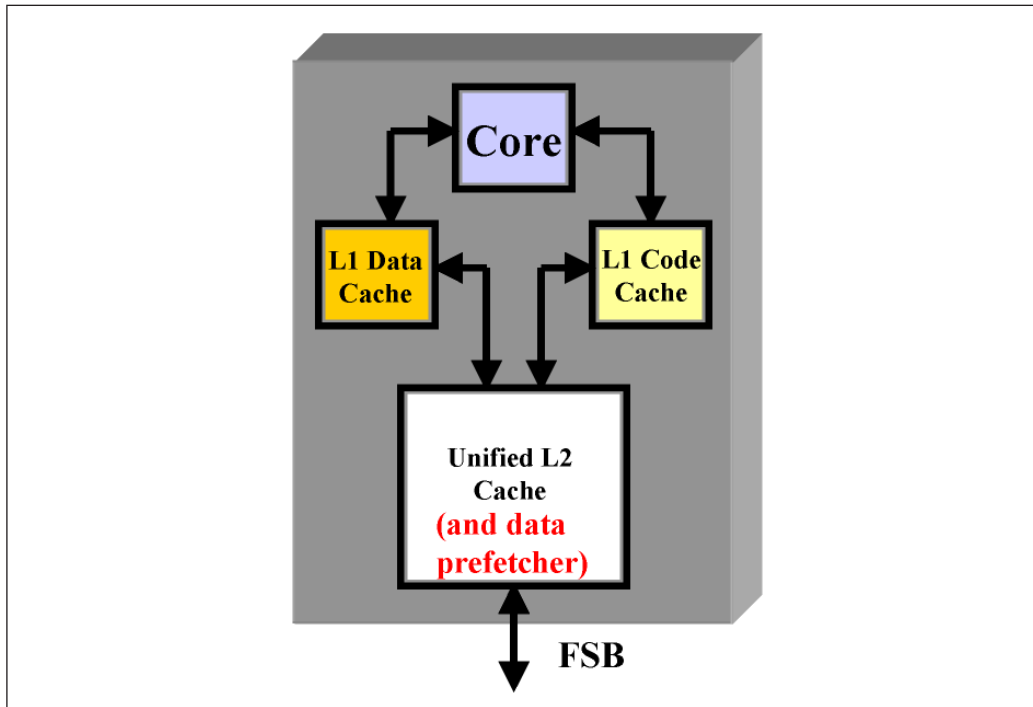
Problem Description

Prior to the Pentium® 4 processor, IA32 processors prefetched instructions but not data. Prefetching instructions before they are actually needed keeps the instruction pipeline fed and hides the delay inherent in using the FSB to access memory. However, the processor had no equivalent “look-ahead” function to anticipate data accesses.

Solution: Add Data Prefetcher

Refer to Figure 1-31 on page 52. The Pentium® 4 added a data prefetcher that attempts to prefetch the data that a program will need before it actually needs it. The data is prefetched into the processor’s L2 Cache.

Figure 1-31: The Pentium® 4 Added a Data Prefetcher



2

Register Set Evolution

100,000 Foot View

Please note that this chapter does not provide a detailed description of each of the registers. Rather, its purpose is to provide a 100,000 foot view of how the registers have evolved over the years. A detailed description of each register can be found in the book.

Introduction

Programs interact with the processor's register set for the following reasons:

- To control its operation.
- To check its current status.
- To manipulate and check the state of data variables.

Starting with the 386 and proceeding to today's Pentium® 4 and Pentium® M processors, this chapter provides an overview of the evolution of the IA32 processor register set.

Don't Forget the Acronym Glossary

The acronym glossary may prove of value in defining some of the terms introduced in this or any other chapter.

Some Commonly-Used Terms

Some terms mentioned in this chapter are described in detail in subsequent chapters. Rather than mentioning them with no description whatsoever, the following is a brief definition of these terms:

- **Real Mode.** IA32 processors are capable of operating in either Real or Protected mode. When in Real Mode, some registers and instructions are unavailable. The currently executing program only has access to the first megabyte (MB) of memory space (00000h through FFFFFh). The data, code and scratchpad (i.e., stack) areas of memory available to the program are each limited to 64KB in size.
- **Protected Mode.** When the processor is placed in Protected Mode, the currently executing program is assigned one of **four privilege levels** by the OS. Privilege level 0 programs are the most privileged and have full access to the instruction set and the register set. Privilege level 3 programs are the least privileged and have limited access to the instruction set and register set. The currently executing program can have access to up to 4GB of memory space. The data, code and scratchpad (i.e., stack) areas of memory available to the program can each be anywhere from one byte to 4GB in size. Programs in memory or on mass storage that are not currently executing are protected from accesses initiated by the currently executing program.
- **Floating-Point (FP) and the FP Unit (FPU).** The FP unit executes FP instructions that operate on numeric data that contains decimal places (as opposed to integer numbers).
- **287 and 387.** The 286 and 386 processors did not incorporate a FPU. Rather, the system board designer had the option of including a separate chip that executed FP instructions forwarded to it by the processor. The 287 FPU could work with either the 286 or 386 processors, while the 387 could only work with the 386.
- **Segment Registers.** These registers define the base address of the data, stack and code areas of memory available to the currently executing program:
 - The memory data segment contains the data items that the currently executing program can access.
 - The memory stack segment of memory can be used by the processor and by the currently executing program to temporarily store information for later retrieval.
 - The memory code segment contains the currently executing program.

Chapter 2: Register Set Evolution

- **Linear Address.** When a memory access must be performed, the processor creates the 32-bit linear memory address by adding an offset to the specified memory segment base address. If **Paging** is disabled, the processor uses the 32-bit linear memory address to access memory. If Paging is enabled, the processor submits the 32-bit linear memory address to the OS-supplied Page Directory and a Page Table in memory to determine where the target memory object really resides in physical memory. The Paging Unit translates the linear address into a physical address in actual memory. The resultant physical memory address is used to access memory.
- **Paging.** See the description of the Linear Address in the previous bullet item.
- **Front Side Bus (FSB).** Refer to Figure 1-1 on page 13 in the book and Figure 1-3 on page 19 in the book. The processor performs transactions on the FSB to communicate with external devices. Likewise, other devices residing on the FSB can communicate with the processor or with each other by performing transactions on the FSB.

The Purpose of the Register Set

The sections that follow introduce the typical register groups implemented within a processor and provides a brief description of each register group's role.

Control Registers

Special control registers are used to control the processor's major operational characteristics. The following are some examples:

- Permits the OS to select whether the processor is currently operating in Real or Protected Mode.
- Permits the OS to select whether or not Paging is currently enabled or disabled.
- Permits the OS to selectively enable or disable some of the processor's features.

Key OS Data Structure Pointer Registers

Some of the processor's registers are used by the OS to identify the location of key data structures in memory. Some examples are:

- The start address of the **Page Directory** in memory.
- The start address of the Global Descriptor Table (**GDT**) in memory.
- The start address of the Local Descriptor Table (**LDT**) in memory.
- The start address of the Interrupt Descriptor Table (**IDT**) in memory.

Instruction Pointer Register Pair

A special register pair (**CS—Code Segment, and EIP—Extended Instruction Pointer**) identifies the memory location from which the processor will fetch the next instruction of the currently executing program.

Debug and Test Registers

A processor may implement special debug registers that permit the programmer to set up memory and/or IO address breakpoints to be monitored by the processor hardware.

The processor may also implement special test registers that permit the testing of the processor's internal memory arrays (e.g., the L1 Caches, the L2 Cache, the Paging-related Translation Lookaside Buffers—TLBs, etc.)

General Purpose Data Registers

Data variables read from memory are placed in the General Purpose Registers (GPRs). The programmer can also create data variables in the processor's GPRs.

When one or more data variables have been read into and/or created in the GPRs, instructions can be executed to:

- Compare the contents of a register with the contents of another register or with a data variable residing in memory.
- Perform arithmetic (e.g., add, subtract, etc.), logical (e.g., or, and, xor, etc.), or shift/rotate operations on the contents of one or more registers and/or memory.

Status Registers

One or more processor status registers track the results of operations performed on the data variables contained in the register set.

FPU Register Set

The Floating Point Unit (FPU) register set consists of control, status and data registers utilized to perform FP math operations (i.e., operations on numbers that contain decimal places).

Error Logging Registers

The processor may contain special registers that are used to log hardware-related errors incurred by the processor's internal units and/or on its FSB.

Interrupt-Related Registers

The processor may implement registers that are used to control and record the status of interrupts delivered to the processor by external hardware devices. These registers may also facilitate the broadcast and reception of Inter-Processor Interrupts (IPIs) messages.

Performance Monitoring Registers

A processor may implement one or more registers that record performance-related information.

Time Stamp Counter (TSC) Register

A processor may implement a register that accurately tracks the amount of time that has elapsed since processor startup (i.e., the removal of Reset).

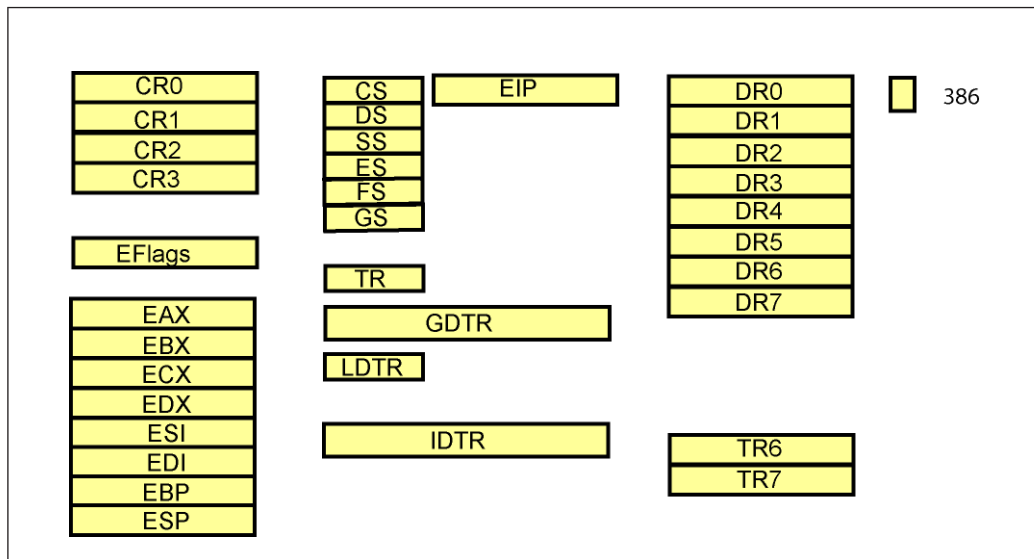
Memory Type and Range Registers (MTRRs)

A processor may implement a register set that divides the processor's overall memory address space into regions and permits the BIOS and/or OS to define the rules of conduct that the processor must follow when performing memory accesses within each region.

Register Set Evolution

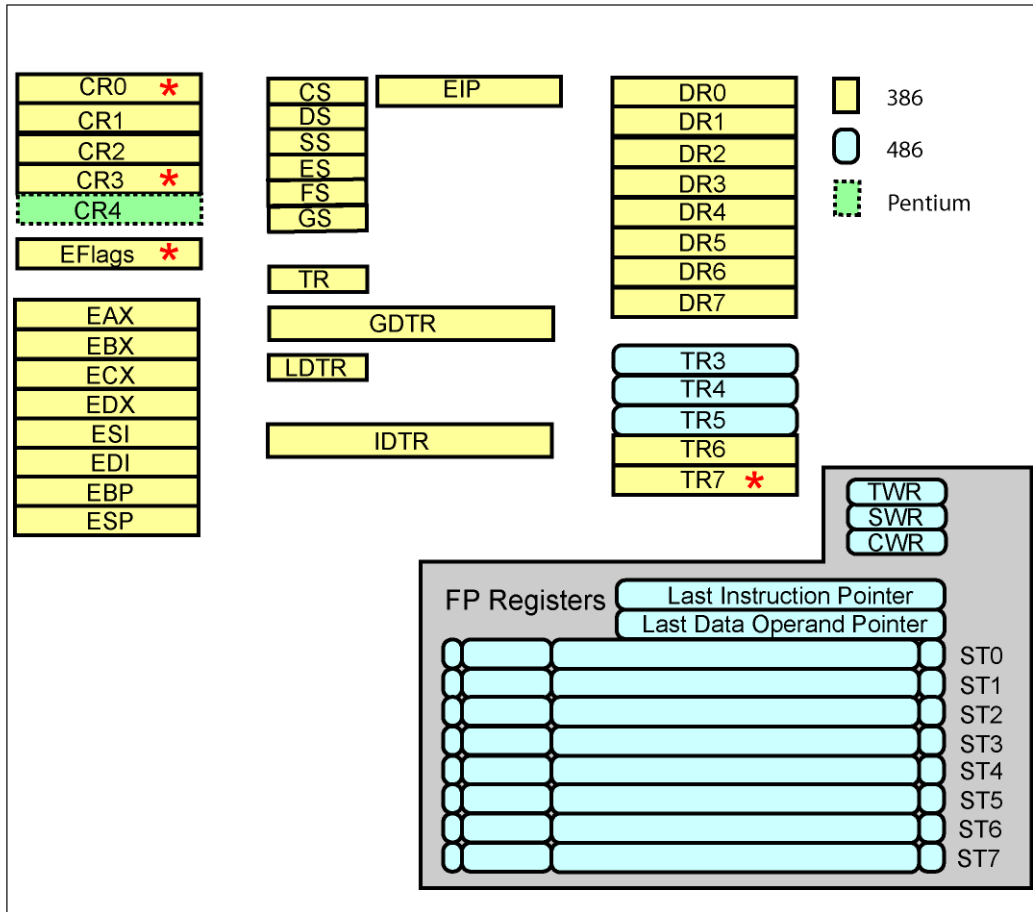
This section provides a fast-forward view that illustrates the growth of the I32 processor register set from the advent of the 386 processor up to the current day Pentium® 4 and Pentium® M processors. Figure 2-1 on page 58 through Figure 2-7 on page 64 illustrate the register set of each processor. The registers are introduced in this chapter and are described in detail in the appropriate sections of the book.

Figure 2-1: 386 Register Set



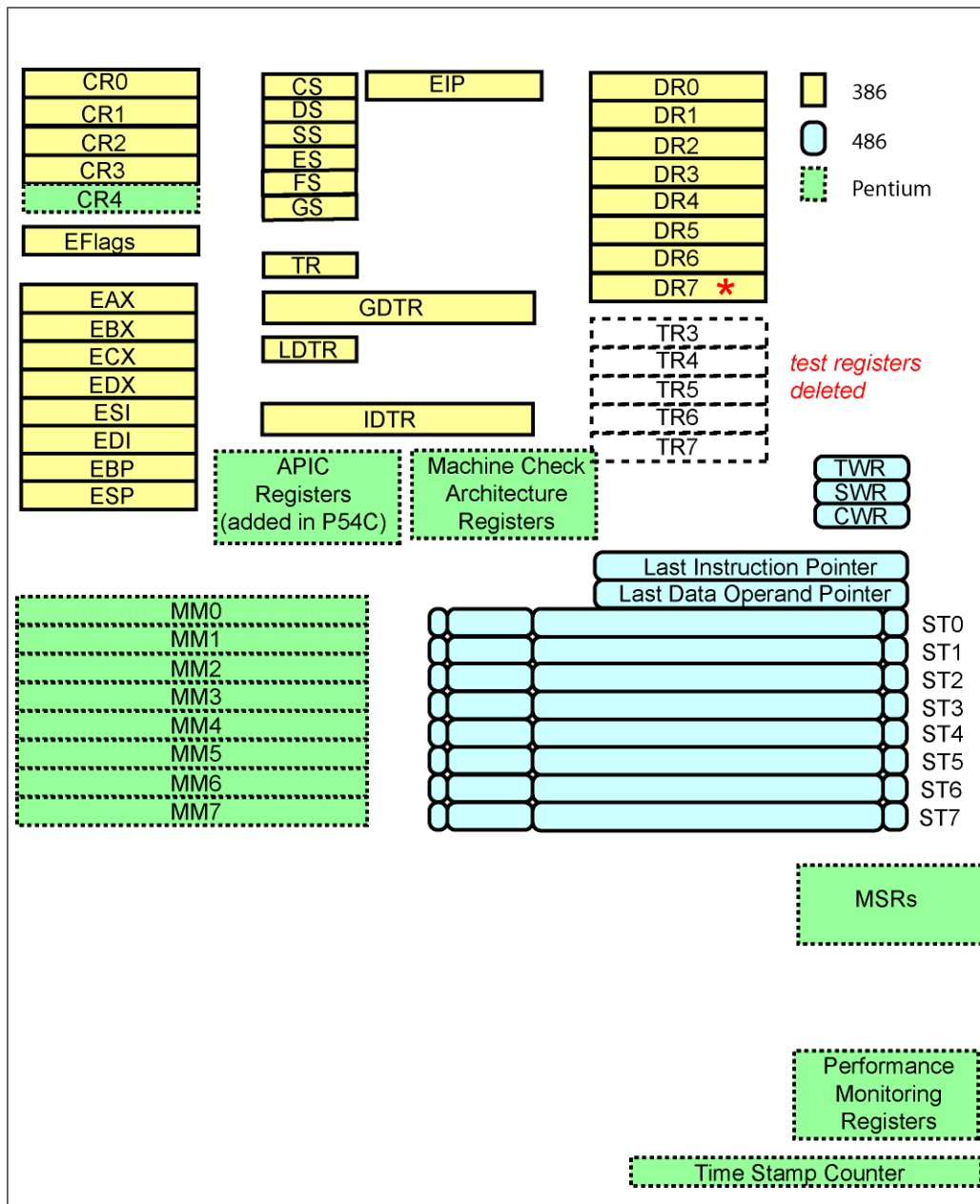
Chapter 2: Register Set Evolution

Figure 2-2: 486 Register Set



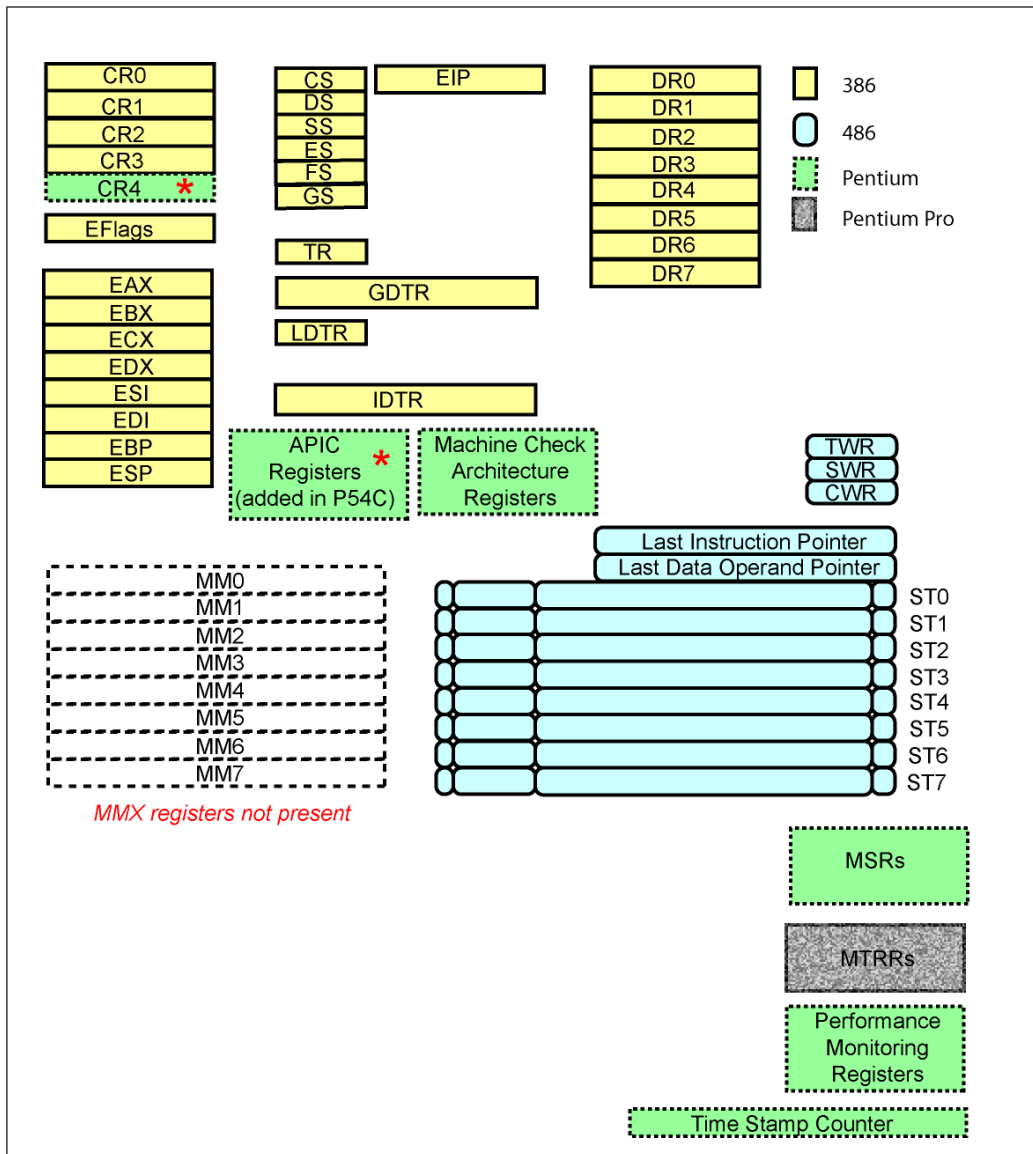
The Unabridged Pentium® 4

Figure 2-3: Pentium® Register Set



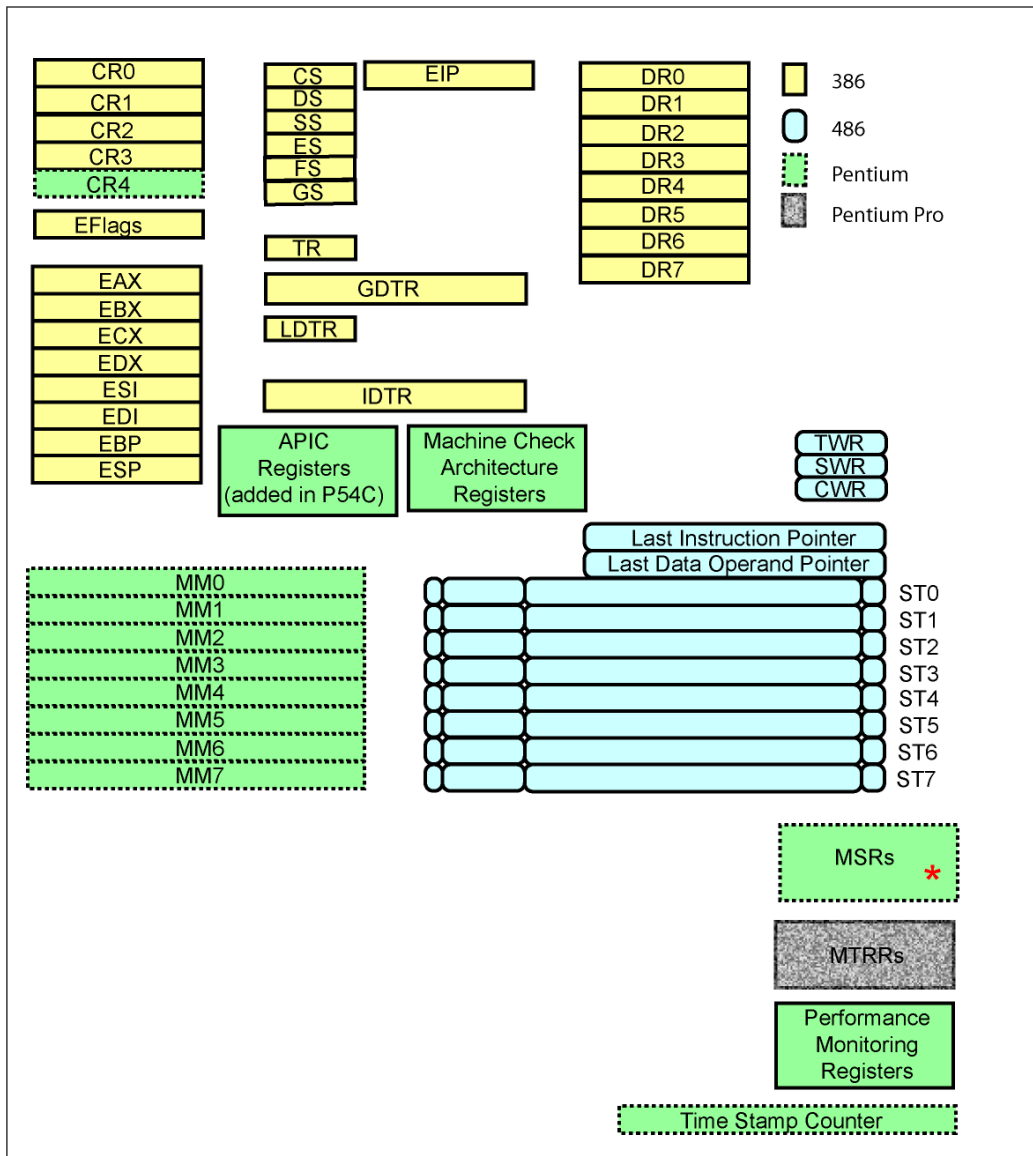
Chapter 2: Register Set Evolution

Figure 2-4: Pentium® Pro Register Set



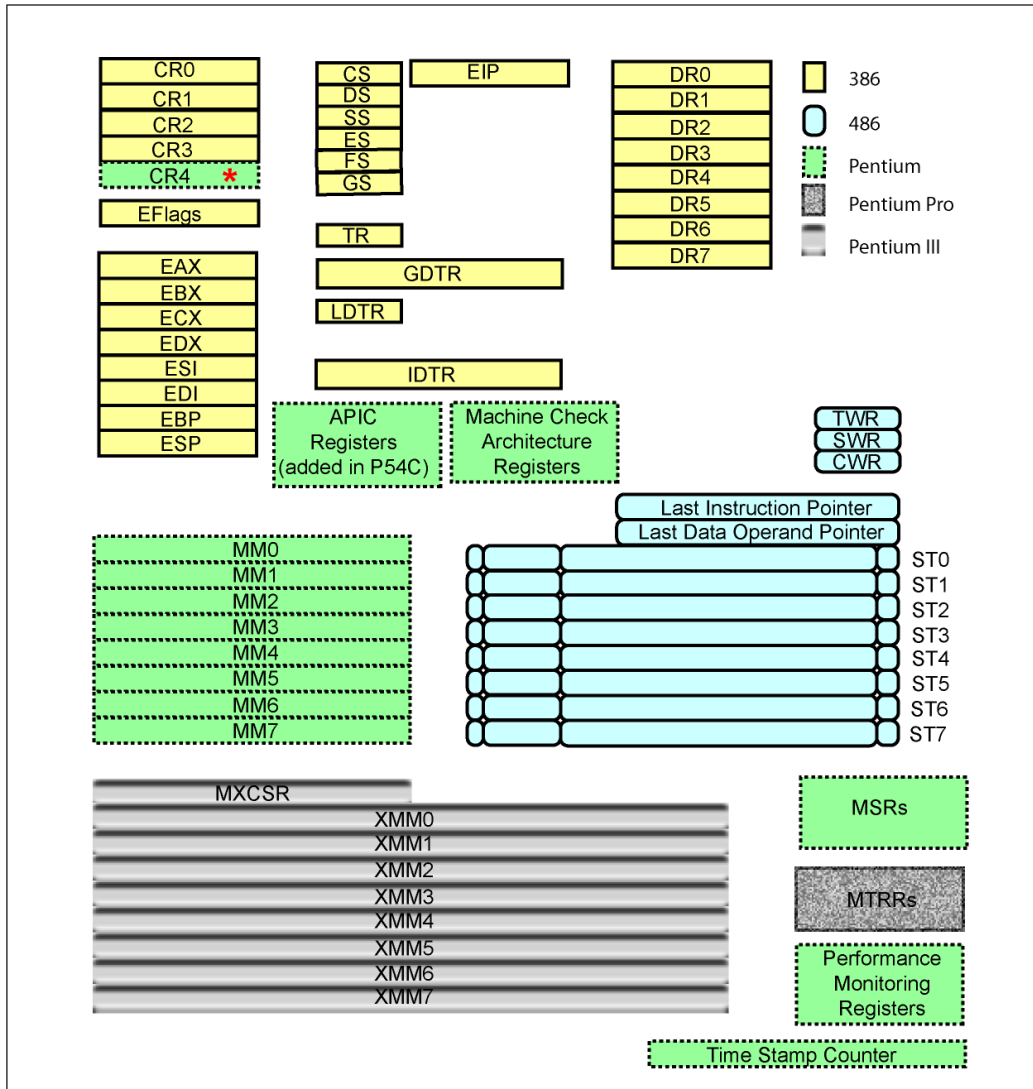
The Unabridged Pentium® 4

Figure 2-5: Pentium® II Register Set



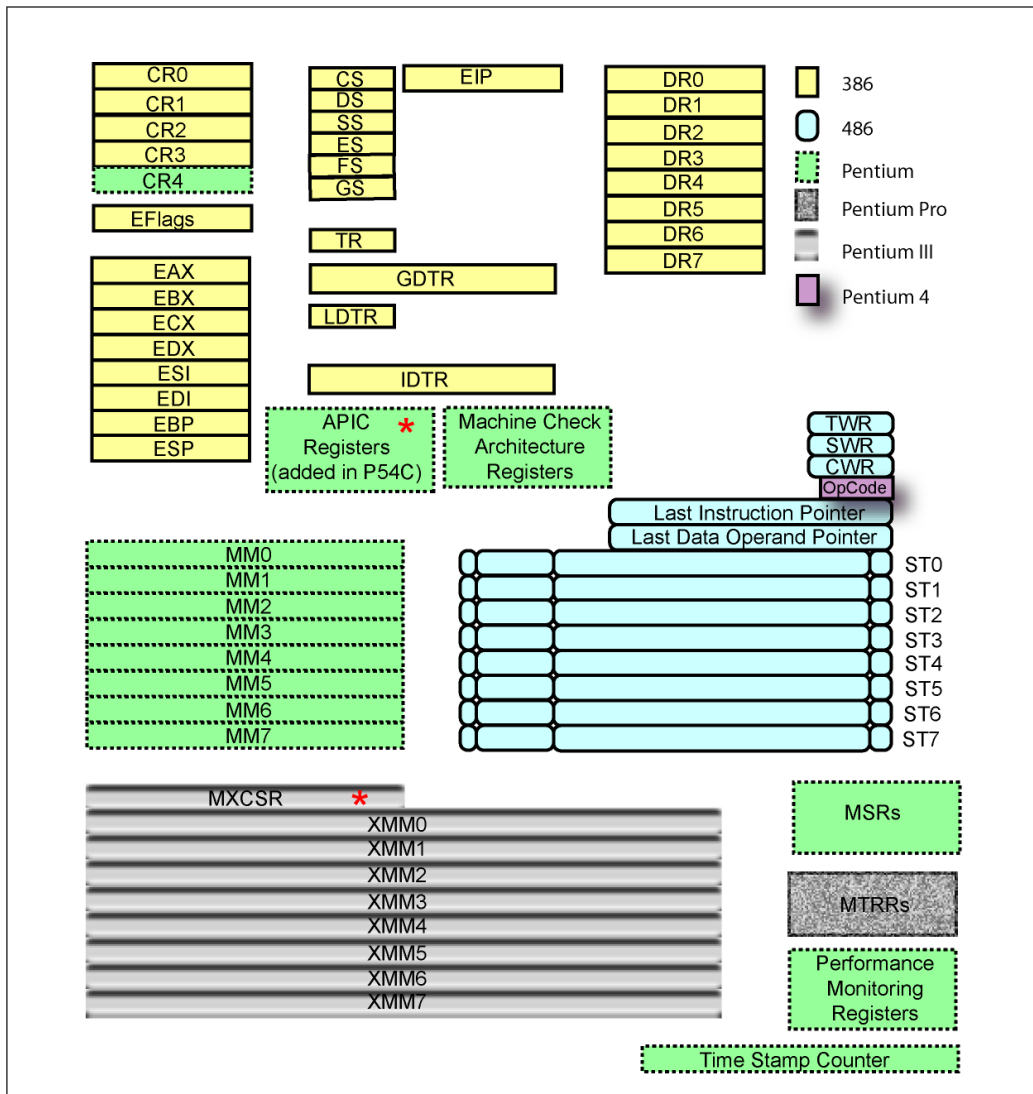
Chapter 2: Register Set Evolution

Figure 2-6: Pentium® III Register Set



The Unabridged Pentium® 4

Figure 2-7: Pentium® 4 and Pentium® M Register Set



The Unabridged Pentium® 4

- Assuming that a x87 FPU coprocessor was present on the system board, to discover whether it is a 287 or 387 style FPU.
- If a FPU is not present, to control whether or not FP instruction emulation is enabled.
- The processor sets the TS bit in CR0 on every task switch and tests it whenever it begins execution of a FP instruction. This permits the saving of the FPU's register set to be delayed until a FP instruction is executed by the new task. Assuming that the TS bit is set, the processor automatically saves the FP data items currently resident in the FP register set and then clears the registers to be used by the new task. A detailed description of FPU operation can be found in "FPU Added On-Die" on page 432 in the book.

Table 2-1: CR0 in the 386

Bit(s)	Name	Description
0	PE	Protected Mode Enable. Defines whether the processor is in Real (0) or Protected Mode (1). A detailed description Real Mode can be found in "386 Real Mode Operation" on page 39 in the book. A detailed description of Protected Mode can be found in the chapters starting with "Protected Mode Introduction" on page 103 in the book.
1	MP	Math Processor Present. This bit is set by software if a FPU (a 287 or 387 numeric coprocessor) is present in the system.
2	EM	FP Emulation Enable. If a FPU is not present, the state of this bit defines whether or not FP instruction emulation is enabled.
3	TS	Task Switch. The processor sets the TS bit in CR0 on every task switch and tests it whenever it begins execution of a FP instruction. This permits the saving of the FPU's register set to be delayed until a FP instruction is executed by the new task. Assuming that the TS bit is set, the processor automatically saves the FP data items currently resident in the FP register set and then clears the registers to be used by the new task. A detailed description of FPU operation can be found in "FPU Added On-Die" on page 432 in the book.

Chapter 2: Register Set Evolution

Table 2-1: CR0 in the 386 (Continued)

Bit(s)	Name	Description
4	ET	Extension Type. Assuming that a FPU is present, the state of this bit indicates whether it is a 287 or 387 style FPU.
31	PG	Paging Enable. Assuming that the processor is in Protected Mode, the state of this bit indicates whether or not Paging is enabled. A detailed description of 386 Paging can be found in “386 Demand Mode Paging” on page 209 in the book.

CR1

CR1 is reserved and has never been implemented in any of the IA32 processors.

CR2

32 bits wide. Refer to Figure 2-10 on page 68. Assuming that the processor is operating in Protected Mode (CR0[PE] = 1) and that Paging is enabled (CR0[PG] = 1), when a memory access is attempted wherein the target page is not currently present in memory (i.e., it resides on a mass storage device, or access to the target page is not permitted), the processor automatically records the offending memory address in CR2.

CR3

32 bits wide. Refer to Figure 2-11 on page 68. After creating the Page Directory data structure in memory, the OS places the upper 20 bits of the 4KB-aligned base address of the Page Directory into CR3. See “386 Demand Mode Paging” on page 209 in the book for a detailed description of Paging.

Figure 2-9: 386 Control Register 0 (CR0)

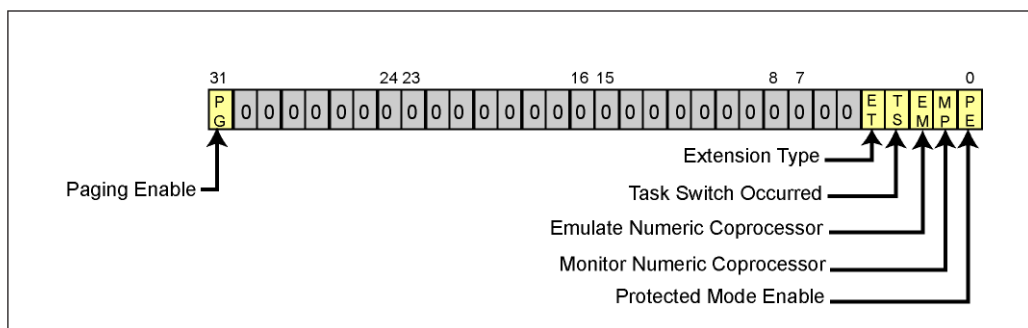


Figure 2-10: Control Register 2 (CR2)

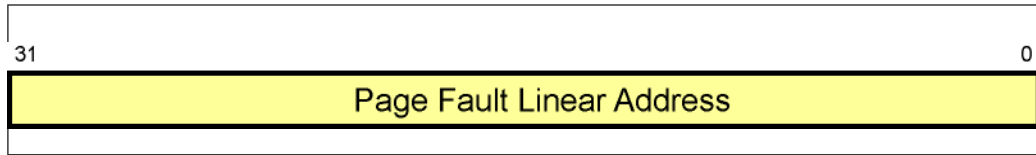
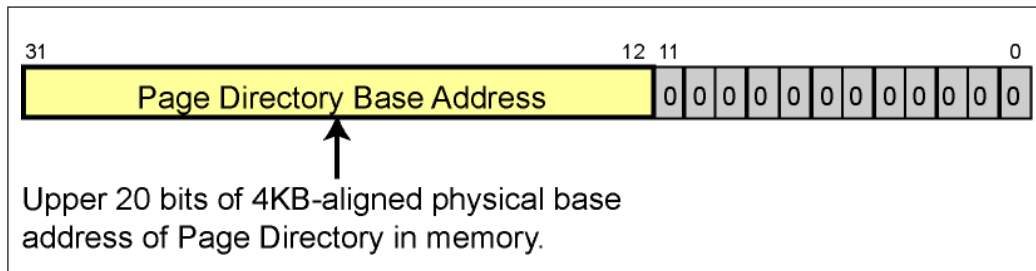


Figure 2-11: 386 Control Register 3 (CR3)



EFlags Register

32 bits wide. Refer to Figure 2-12 on page 69 and Table 2-2 on page 69. The EFlags register serves two purposes:

- EFlags contains a series of status bits that reflect the completion status of the previously executed instruction:
 - Carry Flag (CF).
 - Parity Flag (PF).
 - Aux Carry Flag (AF).
 - Zero Flag (ZF).
 - Sign Flag (SF).
 - Overflow Flag (OF).
- EFlags also contains a series of control bits that control certain aspects of the processor's operation:
 - Trap Flag (TF).
 - Interrupt Flag (IF).
 - Direction Flag (DF).
 - IO Privilege Level (IOPL; this is a two bit field).
 - Nested Task (NT).
 - Resume Flag (RF).
 - Virtual 8086 Mode Flag (VM).

Chapter 2: Register Set Evolution

Figure 2-12: 386 Extended Flags Register (EFlags)

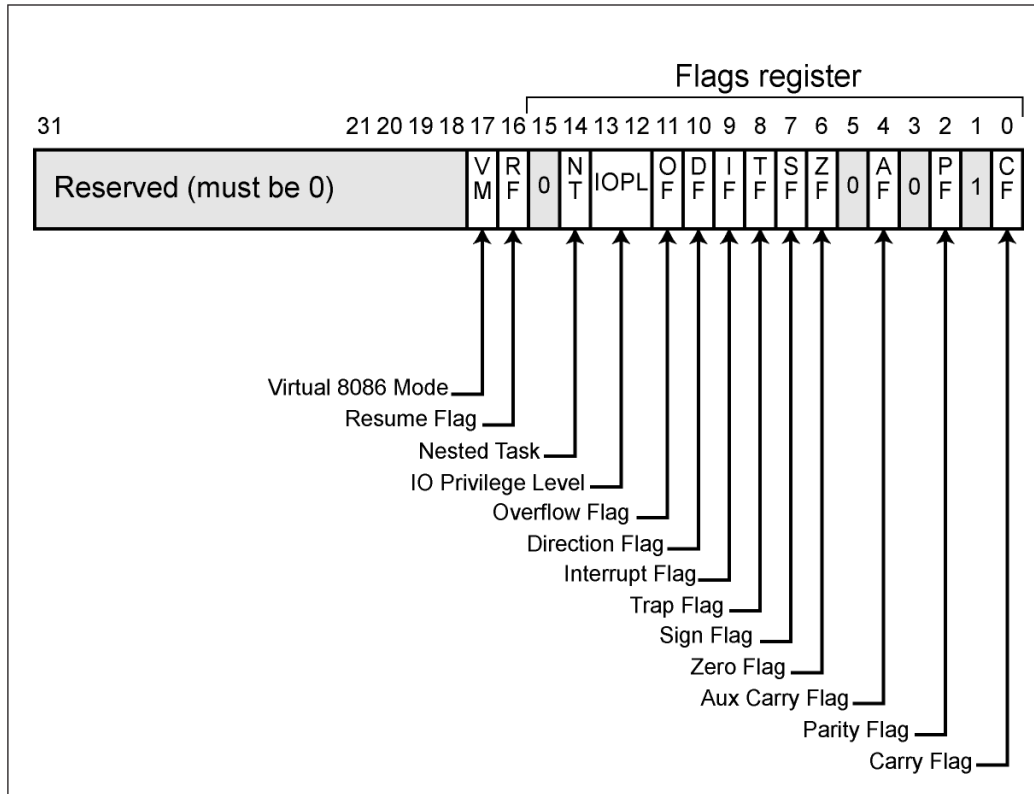


Table 2-2: 386 EFlags Register

Bit(s)	Name	Description
Status Bits		
0	CF	Carry Flag. This bit is set to one if an arithmetic operation generates a carry or a borrow out of the msb of the result; otherwise, it is cleared.
2	PF	Parity Flag. • If the LSB of the result contains an even number of one bits, PF = 1; • Otherwise, PF = 0.

Table 2-2: 386 EFlags Register (Continued)

Bit(s)	Name	Description
4	AF	Aux Carry Flag. This bit is used in binary-coded decimal (BCD) arithmetic. • AF = 1 if a BCD arithmetic operation generates a carry or a borrow out of bit 3 of the result. • Otherwise, AF = 0.
6	ZF	Zero Flag. • ZF = 1 if the result is zero. • Otherwise, ZF = 0.
7	SF	Sign Flag. SF = the value (1 or 0) of the result's msb (the sign bit of a signed integer). • 0 = a positive value. • 1 = a negative value.
11	OF	Overflow Flag. • OF = 1 if the integer result is too large a positive number or too small a negative number (excluding the sign-bit) to fit in the destination operand; it indicates an overflow condition for signed-integer (i.e., two's complement) arithmetic; • otherwise, OF = 0.
Control Bits		
8	TF	Trap Flag. When TF = 1, it enables the processor's single-step mode to facilitate program debug. Prior to the execution of each instruction in the program under debug, a Debug Exception is generated, causing the processor to jump to the program debugger.
9	IF	Interrupt Flag. • 0 = disable recognition of maskable external hardware interrupts. • 1 = enable recognition of maskable external hardware interrupts. See "Interrupts and Exceptions" on page 251 in the book for a more detailed description of interrupts.

Chapter 2: Register Set Evolution

Table 2-2: 386 EFlags Register (Continued)

Bit(s)	Name	Description
10	DF	Direction Flag. The state of the DF bit controls actions taken by the string-oriented instructions (MOVS, CMPS, SCAS, LODS, and STOS): • DF = 1 causes the string instructions to process strings of information in memory starting at the start address and descending through memory (descending from the start address). • DF = 0 causes the string instructions to process strings of information in memory starting at the start address and ascending through memory (ascending from the start address). The STD and CLD instructions set and clear DF, respectively.
13:12	IOPL	IO Privilege Level. In order to successfully execute IO instructions, the Current Privilege Level (CPL) of the currently executing program or task must be numerically <= the IO privilege level. EFlags[IOPL] can only be modified by execution of the POPF and IRET instructions by a program executing at privilege level zero (i.e., the OS kernel). See “IOPL-Sensitive Instructions” on page 345 in the book for a detailed description of IOPL usage.
14	NT	Nested Task. Controls the chaining of interrupted and called tasks: • Set when the current task is linked to (i.e., was called from) another task. • Cleared when the current task is not linked to (was not called from) another task. See “Linked Tasks” on page 201 in the book for a detailed description of nested tasks.

Table 2-2: 386 EFlags Register (Continued)

Bit(s)	Name	Description
16	RF	Resume Flag. If EFlags[TF] = 1, the processor's single-step mode is enabled. Prior to the execution of the next instruction in the program under debug, the processor generates a Debug Exception and jumps to the debug program. Prior to exiting the debug program and returning to the program under debug, the programmer first sets EFlags[RF] to one and then executes the IRETD instruction to return to the program under debug. Upon return to the program under debug, EFlags[RF] = 1 prevents the processor from immediately generating another Debug Exception. Rather, the next instruction in the program under debug is executed before another Debug Exception is generated.
17	VM	Virtual 8086 Mode: • 1 = VM86 mode is enabled. • 0 = VM86 mode is disabled. See "Virtual 8086 Mode" on page 329 in the book for a detailed description of 386 VM86 mode.

General Purpose Registers (GPRs)

Introduction

The 386 implemented the following GPRs (see Figure 2-13 on page 75):

- EAX, EBX, ECX and EDX. These are general purpose data registers, but it should be noted that, when executing some instructions, one or more of them may be used to supply input parameters for the instruction.
- EBP. The Extended Base Pointer register can be used to access data items in memory-based data structures (including the area of memory designated as the stack for the currently executing program).
- ESI and EDI. When executing string operations, these two registers are pre-loaded with the start addresses of the source and destination buffers.
- ESP. The processor uses the register pair consisting of the SS (Stack Segment) register and the Extended Stack Pointer (ESP) register to identify the location currently pointed to in the area of memory designated as the stack for the currently executing program. When the processor is operating in Real Mode, the upper 16 bits of ESP are not used and the lower 16 bits

Chapter 2: Register Set Evolution

(referred to as the SP register) points to the current location in stack memory. When the processor is operating in Protected Mode, the full 32-bit ESP points to the current location in stack memory.

EAX, EBX, ECX and EDX Registers

Refer to Figure 2-13 on page 75. Each of these four registers can hold 32 bits (four bytes) of information. As an example, the following instruction will move the 32-bit value 12345678h into the EAX register:

```
mov eax,12345678 ;move the value 12345678h into eax
```

In addition, the programmer can individually refer to each of the 8- or 16-bit registers that make up the lower half of each of these registers. As an example, the 16-bit AX register actually consists of two 8-bit registers: the AL register (the lower half of AX register) and the AH register (the upper half of the AX register). The same is true of the BX, CX and DX registers. If the programmer wants to move only one byte of information, he or she would specify one of the 8-bit registers as the source or target register (rather than a 16-bit register). In the following example, the programmer wants to write the value 02h into IO location 60h:

```
mov al,02      ;move value 02h into al
out 60,al      ;write al contents to i/o port 60h
```

In the example, the programmer first moves the value 02h into the 8-bit AL register and then performs an IO write instruction to write the contents of the AL register to IO location 60h. This will cause an IO write transaction to be performed on the FSB with address 000060h on the address bus and 02h on the lower part of the data bus (D[7:0]#). In general, the AX, BX, CX and DX registers are used for the following purposes:

- to temporarily hold values to be used in calculations;
- to receive data being read from an external memory or IO location;
- to provide the data to be written to an external memory or IO location;
- to hold values to be used in calculating a memory address.

EBP Register

Although available for general use by the programmer, the 32-bit Extended Base Pointer (EBP) register is frequently used in forming a memory address. The programmer places the start address of a table of data items in memory (often referred to as a data structure) into the EBP register. A move instruction is then executed that adds an index value to this base address to form the exact

address of a data item in the table in memory. This is known as indexing into a table. The EBX register can also be used in the same manner. The EBP register is often used to access the stack without affecting the contents of the ESP register.

Index Registers

The two index registers are:

- The Source index (ESI) register.
- The Destination index (EDI) register.

These two registers are frequently used for operations on strings of data in memory. This is an operation that is performed on a string of locations, rather than just one.

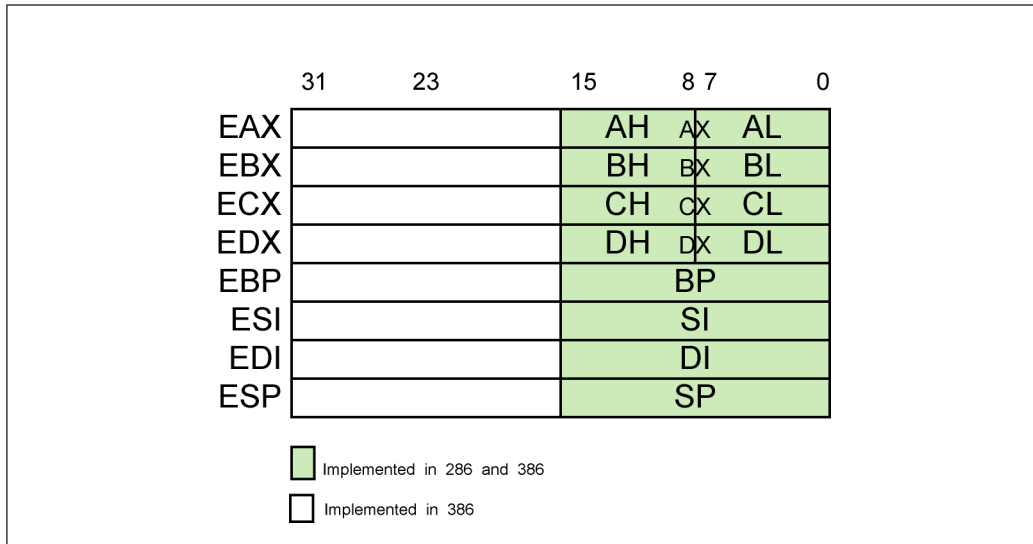
A classic example of a string operation would be the move string operation where the programmer wants to perform a series of memory reads and writes to transfer a block of data from one area of memory to another. To accomplish this, the programmer would code the following instructions:

```
    mov esi,xxxx    ;esi=start address of source buffer
    mov edi,yyyy    ;edi=destination buffer start address
    mov ecx,zzzz    ;cx = the number bytes to move
rep movsb           ;move string of bytes
                   ;repeat until done
```

In this example, the programmer places the start memory address of the area of memory containing the source data into the ESI register and the start memory address of the destination area in memory into the EDI register. The number of bytes to move is placed into ECX, which acts as the count register in this context. The programmer then executes the Repeat Move String of Bytes (REP MOVSB) instruction. When executed, this instruction performs a memory read from the memory address specified in the ESI register. The byte received from this location is then written to the memory location specified in the EDI register. The ECX register is then decremented and the ESI and EDI registers are incremented. When the count in the ECX register is exhausted, all of the bytes have been moved and the REP MOVSB instruction has completed execution. The next instruction is then executed. If the count is not exhausted, however, the REP MOVSB is repeated as many times as necessary until the block move is completed.

Chapter 2: Register Set Evolution

Figure 2-13: The General Purpose Registers (GPRs)



Segment Registers

Refer to Figure 2-14 on page 76 and Figure 2-15 on page 77. The segment registers are used to identify the following regions (or segments) of memory:

- The data segments that hold the data variables and/or constants that the currently executing program operates upon.
- The stack segment that is used as scratchpad memory by the currently executing program (and, for certain operations, by the processor).
- The code segment that contains the currently executing program.

Real Mode Usage

If the processor is operating in Real Mode, each 16-bit segment register contains the base address of the respective memory region and the length of the segment = 64KB. See “386 Real Mode Operation” on page 39 in the book for a detailed description of Real Mode operation.

Protected Mode Usage

If the processor is operating in Protected Mode, the 16-bit value in each segment register identifies an entry in either the Global or Local Descriptor Table. The

table entry selected by the segment register's contents identifies the respective segment's start address, its length, its access rights, and identifies whether or not it's a special OS, kernel-related segment. See the chapters starting with "Protected Mode Introduction" on page 103 in the book for a detailed description of Protected Mode operation.

DS, ES, FS and GS

The Data Segment registers identify the regions of memory that hold the data variables and/or constants that the currently executing program operates upon.

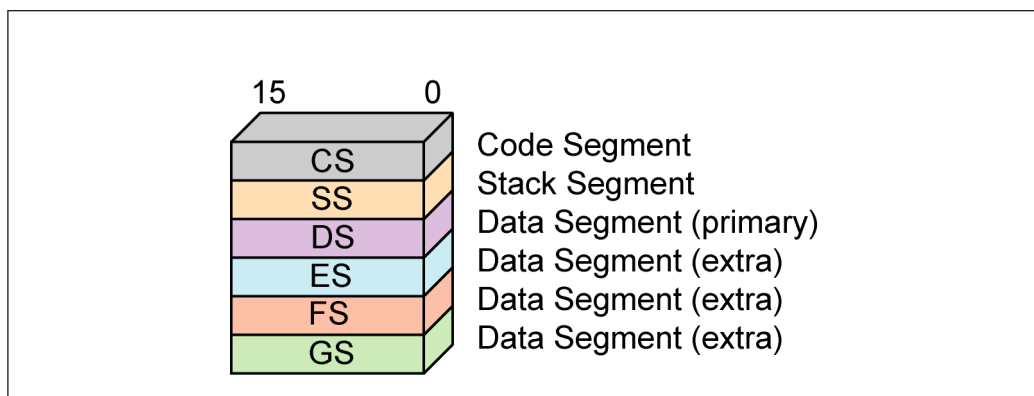
CS

The Code Segment register identifies the start address of the area of memory that contains the currently executing program. Also see "Extended Instruction Pointer (EIP) Register" on page 78.

SS

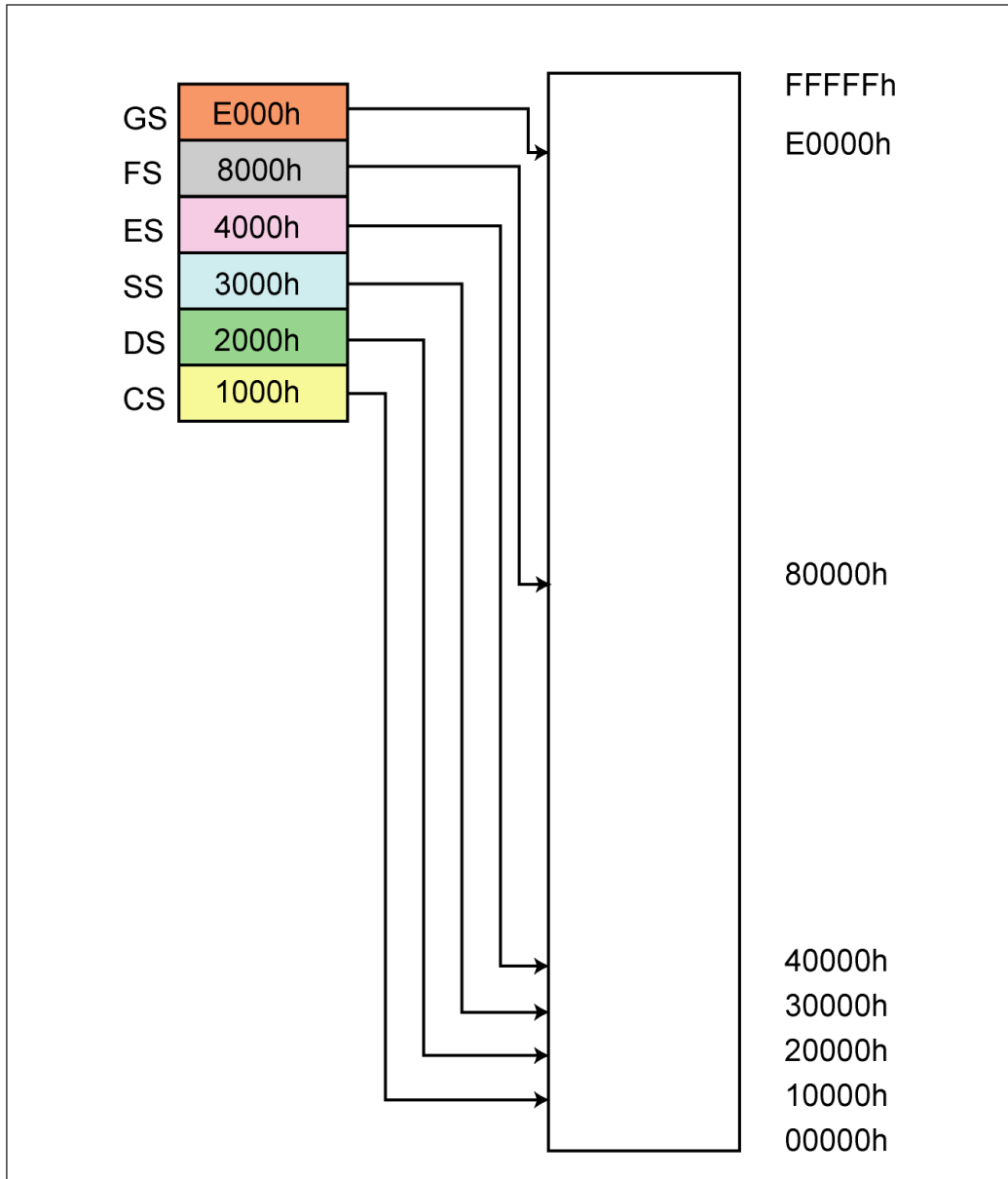
The Stack Segment register identifies the start address of the area of memory that is used as scratchpad memory by the processor and by the currently executing program. See "Accessing the Stack Segment" on page 76 in the book and "Selecting and Accessing a Stack Segment" on page 161 in the book for a detailed description of stack operation.

Figure 2-14: The Segment Registers



Chapter 2: Register Set Evolution

Figure 2-15: Segment Registers Identify Memory Areas Associated with Executing Program



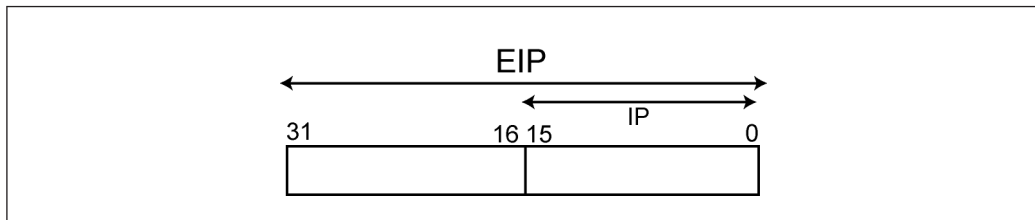
Extended Instruction Pointer (EIP) Register

Refer to Figure 2-16 on page 78. The Code Segment (CS) register identifies the start address of the area of memory that contains the currently executing program, while the 32-bit EIP register identifies the exact location within the code segment that the next instruction of the currently executing program will be fetched from.

When the processor is operating in Real Mode, the 16-bit value in the IP portion of the EIP register identifies the location (in the Code Segment area of memory) from which the next instruction will be fetched (and the upper 16 bits of EIP are not used).

When the processor is operating in Protected Mode, the full 32-bit value in the 32-bit EIP register identifies the location (in the Code Segment memory area) from which the next instruction will be fetched.

Figure 2-16: EIP Register



Task Register

What Is a TSS?

A multitasking OS creates a special data structure in memory for each task (i.e., program) that will be executed. The processor and/or OS uses this data structure (the Task State Segment, or TSS; see Figure 2-17 on page 80) in the following manner:

- Before the task is executed for the first time, the OS builds the TSS data structure in memory. Among other items, the TSS contains the initial values to be placed in each of the processor's registers when the task begins execution for the first time.

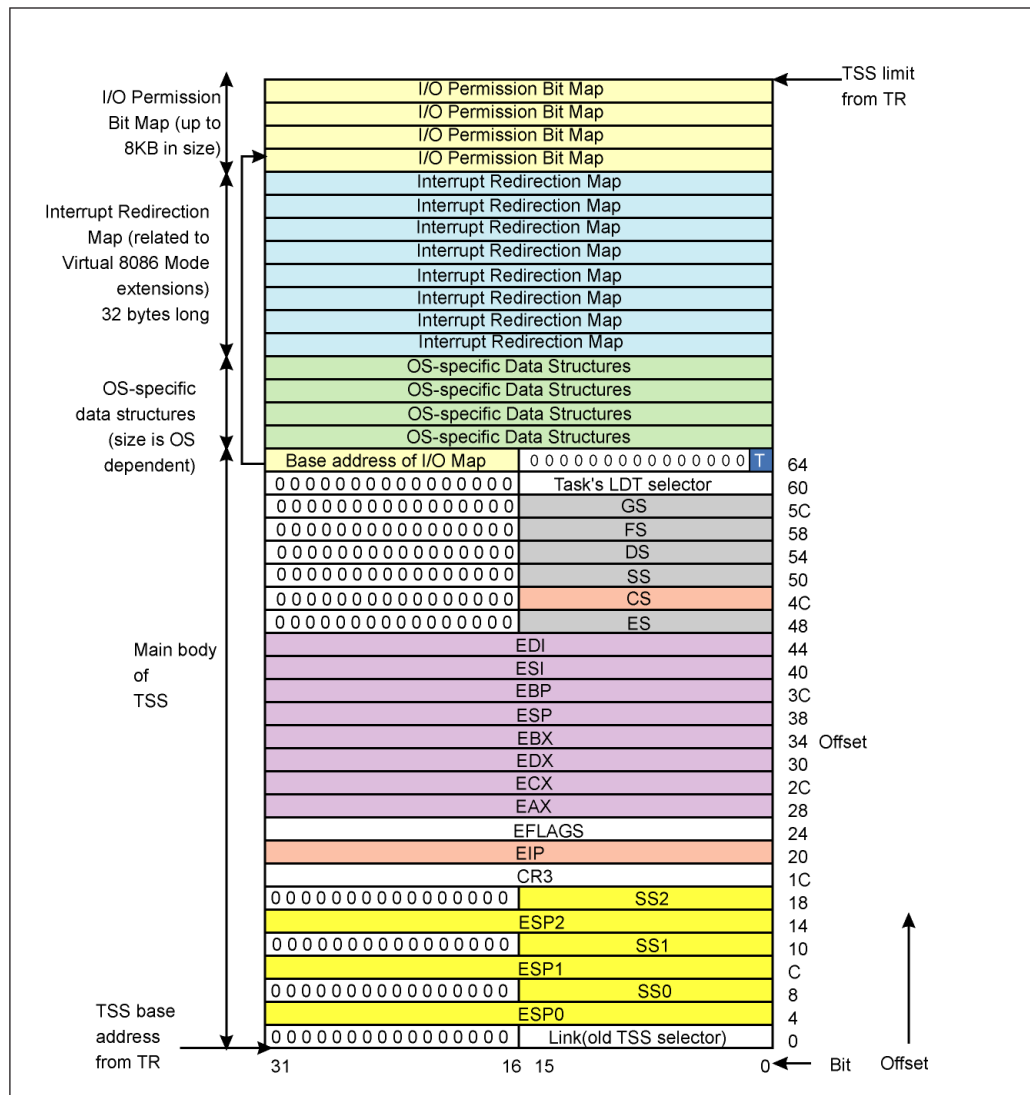
Chapter 2: Register Set Evolution

- When the processor begins executing the task for the first time, it automatically reads the initial register values into its register set from the respective locations in the TSS.
- Whenever the task is suspended (e.g., when the task's slice of processor execution time has expired), the processor automatically stores the current contents of most of its register set into the respective locations of the TSS.
- Whenever execution of a previously suspended task is resumed, the processor automatically reads the values stored in the TSS into its register set.
- Some additional task-related information is also stored in the TSS.

The Purpose of the Task Register

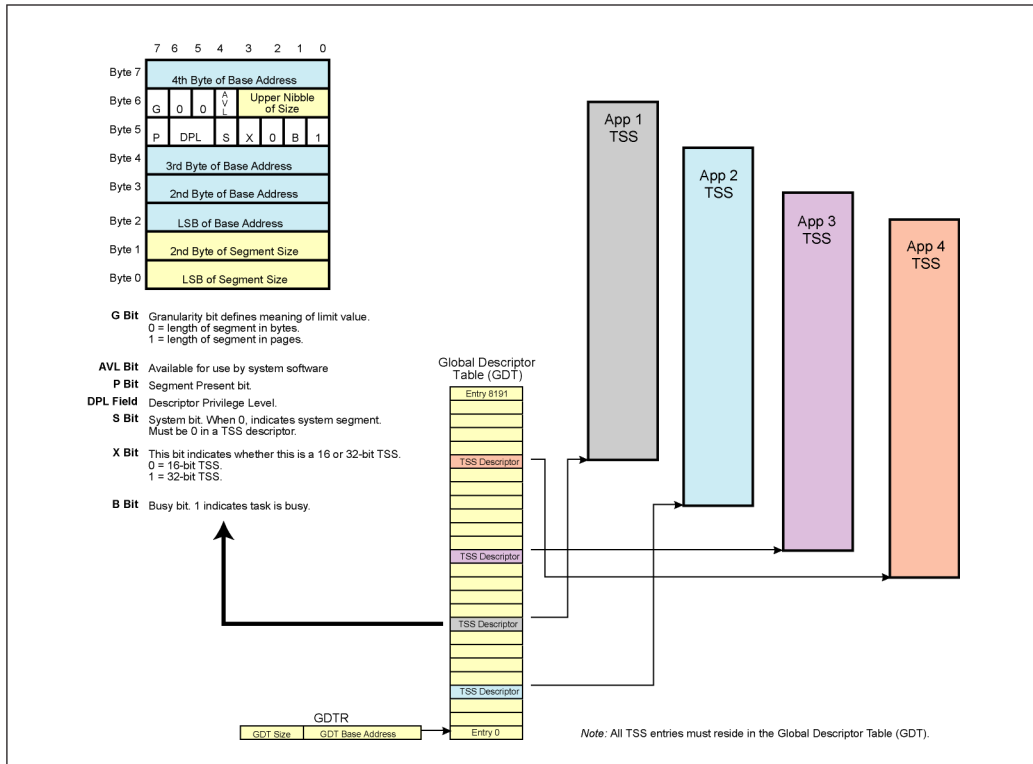
Refer to Figure 2-18 on page 81. In order to save or restore its register set to or from the currently executing task's TSS, the processor would have to know where the TSS related to the currently executing program resides in memory. The TR tells the processor where the task's TSS resides in memory. The 16-bit TR identifies which entry in the memory-based GDT (Global Descriptor Table) contains the start address and length of the TSS for the currently executing task.

Figure 2-17: Task State Segment (TSS)



Chapter 2: Register Set Evolution

Figure 2-18: The Task Register Identifies the TSS Associated with the Currently Executing Task



GDTR and LDTR

See the chapters starting with “Protected Mode Introduction” on page 103 in the book for a detailed description of Protected Mode operation.

Refer to Figure 2-19 on page 82. The Global Descriptor Table Register (GDTR) is only used when the processor is operating in Protected Mode. It points to the GDT in memory. The GDT contains the following:

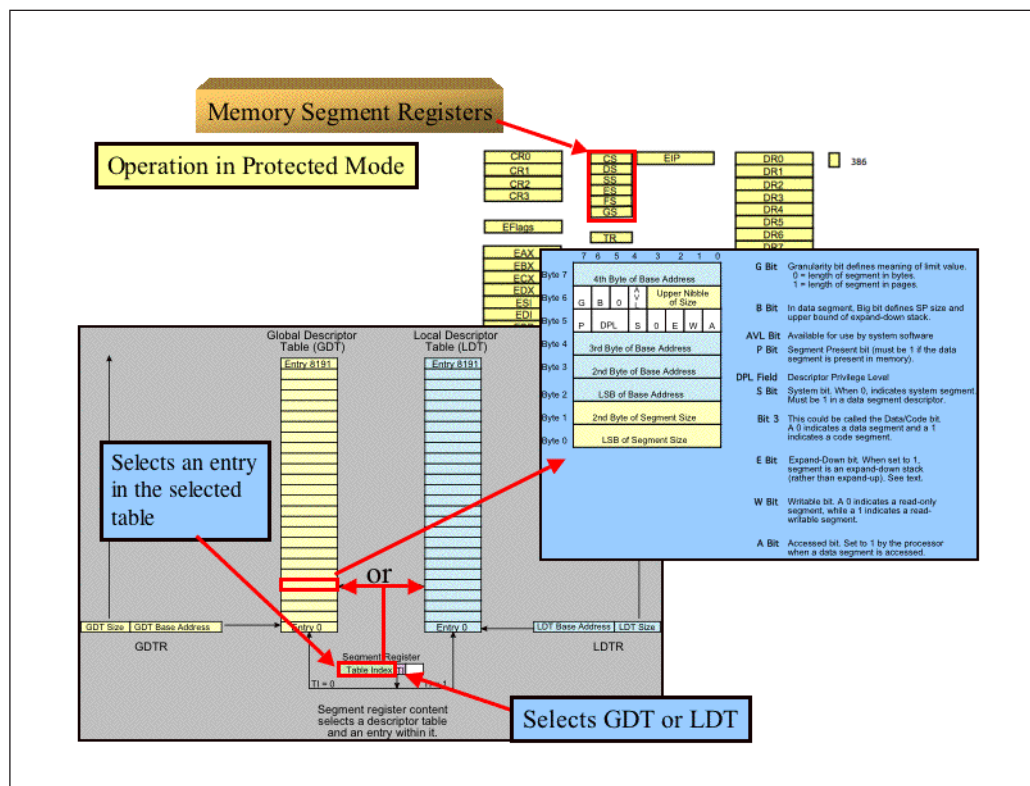
- The segment descriptors that define the start address, length, and access rights for memory segments that are accessible by multiple tasks.
- Descriptors that point to the TSS data structures associated with each of the tasks that the OS is currently executing.
- Descriptors that point to Local Descriptor Tables (LDTs) associated with each of the tasks that the OS is currently executing.

The Unabridged Pentium® 4

The Local Descriptor Table Register (LDTR) contains the segment descriptors that define the start address, length, and access rights for the segments that are accessible only by the currently executing task.

When the currently executing program loads a new 16-bit value into one of the segment registers, that value selects either the GDT or LDT as well as an entry in the selected table. The processor then automatically reads the indicated entry from the selected table and that entry's contents defines the memory segment's start address, length, and access rights.

Figure 2-19: GDTR and LDTR



IDTR

See “Interrupts and Exceptions” on page 251 in the book for a detailed description of interrupts.

Hardware Interrupts

When a hardware device requires servicing, it is most frequently designed to alert the processor by generating an interrupt. The interrupt provides the processor with an 8-bit value that identifies the device requiring service. The OS builds and maintains a 256-entry Interrupt Table in memory and each entry contains a pointer to the interrupt handler routine associated with a specific device. When an interrupt is delivered to the processor, the processor uses the 8-bit value (the interrupt vector) to select an entry in the Interrupt Table. The processor suspends execution of the currently executing program and jumps to the device-specific interrupt service routine pointed to by the selected table entry. Execution of the interrupt service routine (the ISR; a subset of the interrupting device’s device driver) services the device and clears the pending interrupt request.

Software Exceptions and Interrupts

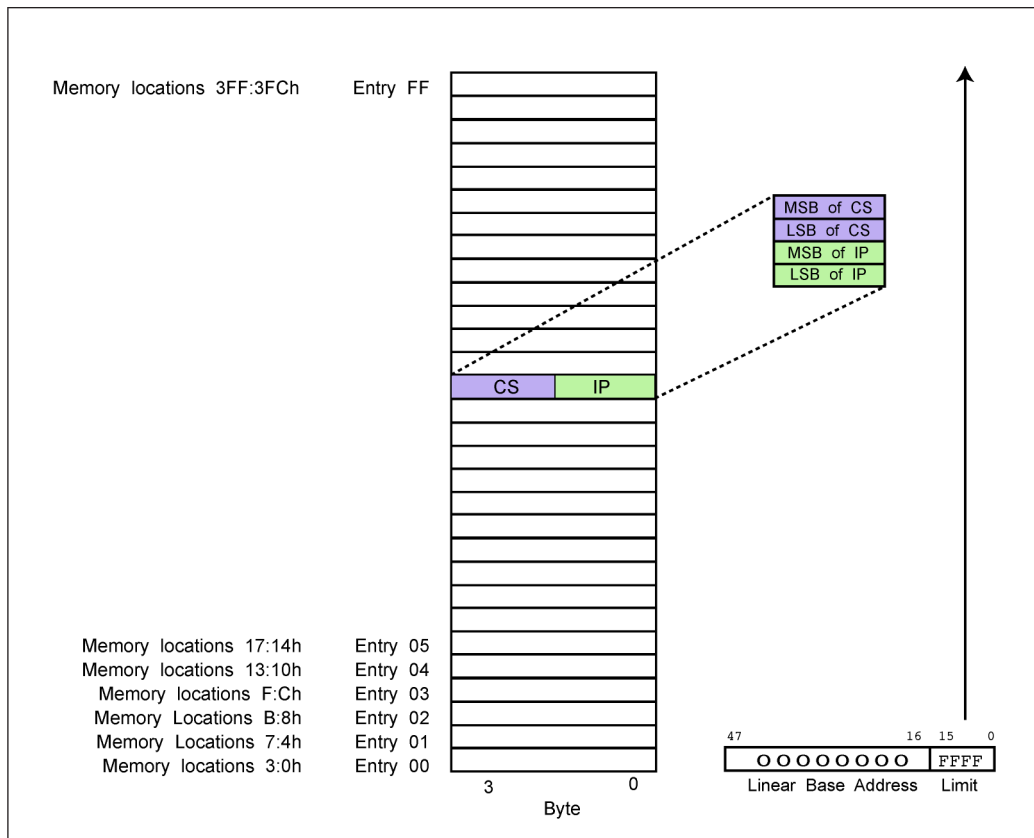
When the processor attempts to execute an instruction and incurs an error in doing so, it generates an internal interrupt referred to as a software exception. The type of error condition defines the type of software exception and each type of exception is identified by an 8-bit value. The processor uses the 8-bit value to select an entry in the Interrupt Table. The processor suspends execution of the currently executing program and jumps to the exception-specific routine pointed to by the selected table entry. Where possible, the code that comprises the selected exception handler attempts to gracefully fix the error so the processor can resume execution of the interrupted program.

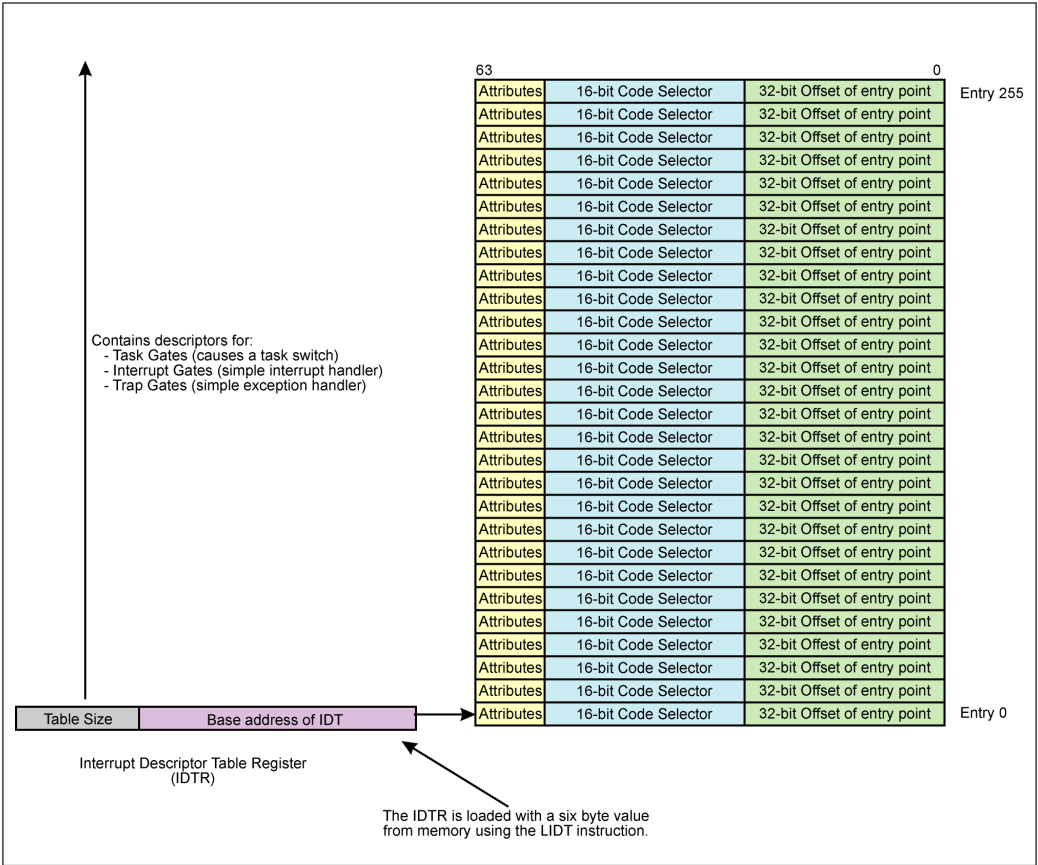
The currently executing program can use the software interrupt instruction, `INT nn` (where `nn` = an entry in the Interrupt Table), to call a routine to be executed. Upon execution of the `INT` instruction, the processor suspends execution of the currently executing program, saves (pushes) the address of the instruction immediately following the `INT` instruction into stack memory, uses the `nn` value to select an entry in the IDT (Interrupt Descriptor Table), and begins fetching and executing the program pointed to by that entry. The last instruction in the called program is the `IRET` instruction. When executed, it causes the processor to read (i.e., `pop`) the previously stored memory address from stack memory into the `CS:EIP` register pair and the processor resumes execution of the calling program.

IDTR Points To the Interrupt Table

In order for this to work, the processor obviously must know the start address of the Interrupt Descriptor Table (the IDT). After creating the IDT in memory, the OS kernel programs the base address and length of this table into the IDTR (Interrupt Descriptor Table Register). Figure 2-20 on page 84 shows the IDT structure when the processor is operating in Real mode, while Figure 2-21 on page 85 shows the structure when the processor is operating in Protected Mode.

Figure 2-20: The IDT in Real Mode





See “The Debug Registers” on page 375 in the book for a detailed description of the Debug registers.

Refer to Figure 2-22 on page 86. The debug registers provide the ability to set both code and data breakpoints to facilitate program debugging:

- DR7 is the Debug Control Register and is used to define and enable up to four breakpoint monitor addresses. In addition, bits in DR7 may be used to define a breakpoint as local to a particular task or as global in nature.

The Unabridged Pentium® 4

- DR0 through DR3 are used to specify the memory address at which each of up to four breakpoints is set.
- DR6 is the Debug Status Register that records which of the four addresses had a breakpoint match.

In the case of a code breakpoint, the processor monitors every instruction fetch for a match on the specified memory address. When a match is detected a debug exception occurs, causing the processor to jump to the debug program which then displays the processor's register set.

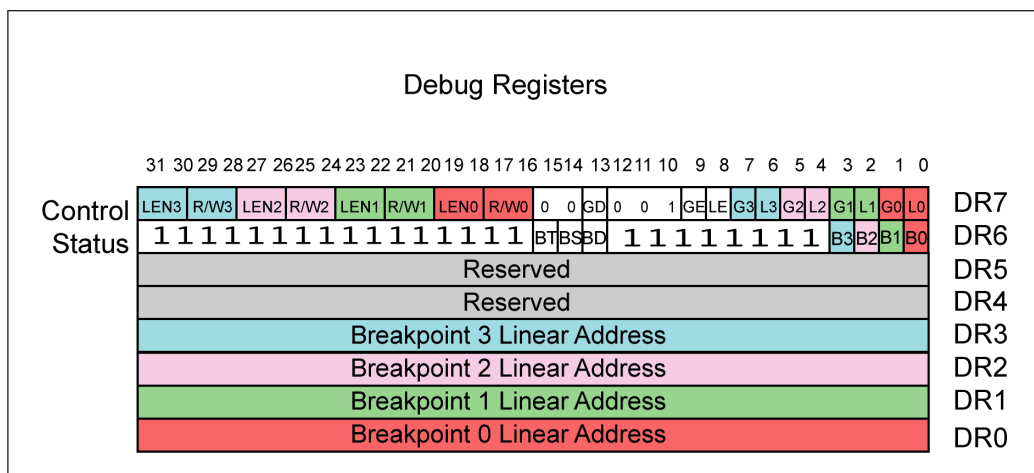
In the case of a data access breakpoint, the processor monitors for a data read from or a data write to the specified memory address defined by the programmer. When a match is detected a debug exception occurs, causing the processor to jump to the debug program which then displays the processor's register set. When defining a data access breakpoint, the programmer may also specify the size of the access (one byte, two bytes, or four bytes) and whether to monitor for a read or a write access.

If enabled, a breakpoint may take one of three forms:

- Break on instruction execution only.
- Break on data writes only.
- Break on data reads or writes, but not on instruction reads.

DR4 and DR5 are not implemented.

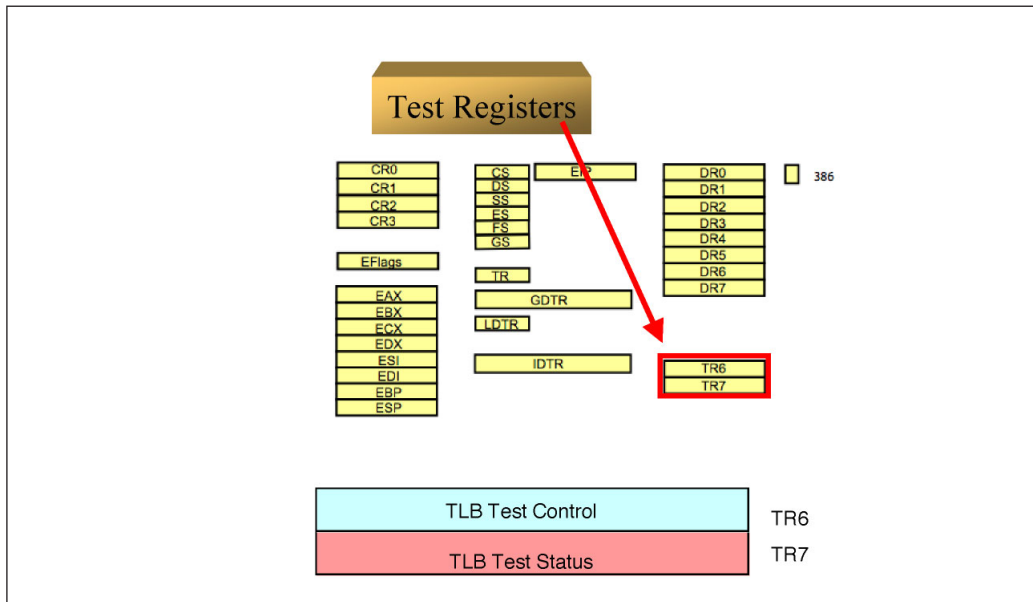
Figure 2-22: Debug Registers



Test Registers

TR6 and TR7 were included in the 386 to permit testing of the small, special-purpose cache within the processor that cached copies of the most recently accessed entries from the Page Tables in memory. The test registers are processor design-specific. Whether or not they are implemented and, if so, how they are implemented can vary from processor to processor. They were implemented in the 386 and 486 but were eliminated starting with the Pentium® processor. They are not part of the IA32 processor architecture specification. For this reason, the author has opted not to include a detailed description of these registers.

Figure 2-23: 386 Test Registers

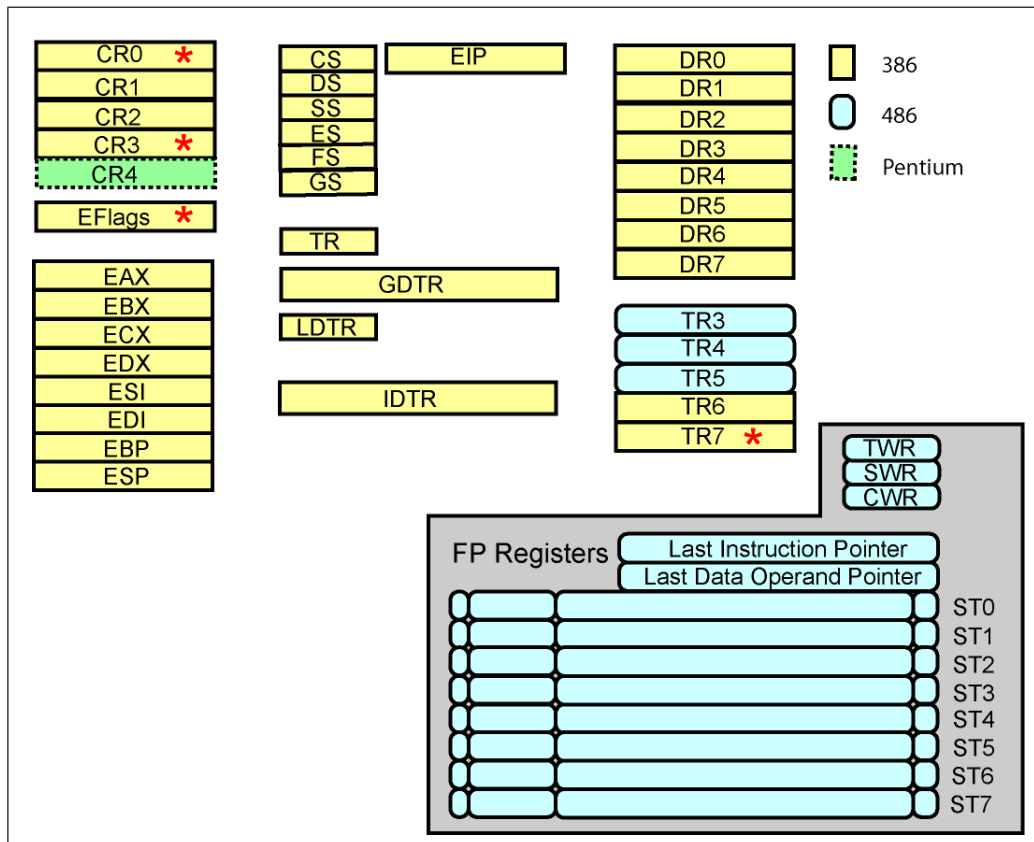


The 486 Register Set

Figure 2-24 on page 88 illustrates the 486's register set. Register set changes are briefly described in the subsections that follow and are indicated in the illustration in the following manner:

- Registers that were present in the 386 wherein one or more new bits were added in the 486 are indicated via an asterisk (*).
- New registers that were added in the 486 are:
 - Test Registers 3, 4, and 5.
 - The FPU register set.
- Although CR4 first appeared in the Pentium® processor, it also appeared in the later versions of the 486.

Figure 2-24: 486 Register Set



FPU Register Set

See “FPU Added On-Die” on page 432 in the book for more information on FPU operation.

The early versions of the 486 processor did not incorporate the Floating Point Unit (FPU). The later versions, however, all implemented an integrated FPU incorporating the FPU register set. This register set consists of:

- The eight data registers, ST[7:0].
- The Last FP Data Operand Pointer and Last FP Instruction Pointer registers. These two registers contain the memory address of the last computational (i.e., non-control) FP instruction executed and, if that instruction referenced a numeric operand in memory, the memory address of that operand.
- The Tag Word Register (TWR). This 16-bit register is divided into eight fields, each of which is two bits wide. Each field is associated with one of the eight data registers and indicates the current contents of that register:
- The Status Word Register (SWR). The bits in this 16-bit register reflect the current status of the FPU.
- The Control Word Register (CWR). This 16-bit register controls the FPU’s precision as well as the rounding method that it uses. In addition it contains six mask bits that permit the programmer to select which types of FP exceptions are enabled.

New Test Registers

The 486 added Test Registers TR3, TR4, and TR5 to facilitate the testing of the on-chip, level 1 unified cache. In addition, the PCD, PWT and LRU bits were added to TR7 to facilitate testing of the on-die, level 1 unified cache.

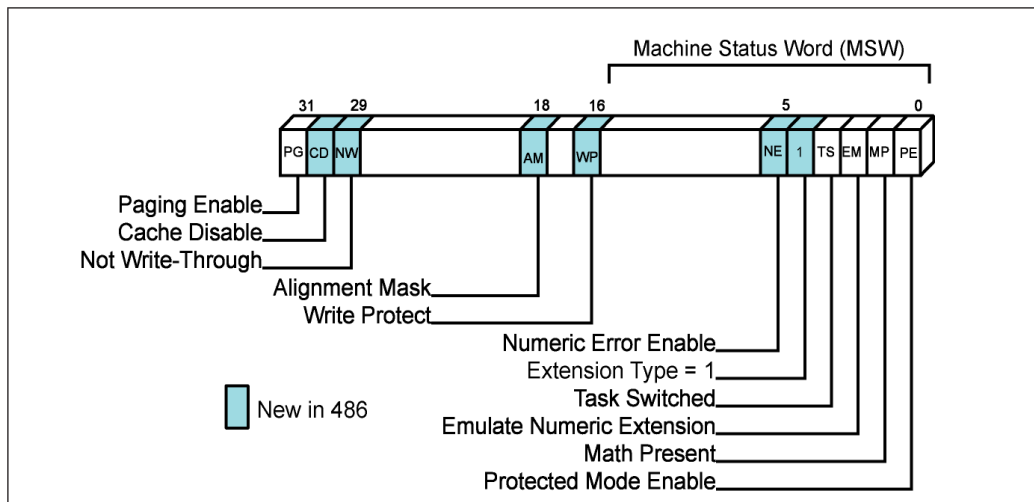
The test registers are processor design-specific. Whether or not they are implemented and, if so, how they are implemented can vary from processor to processor. They were implemented in the 386 and 486 but were eliminated starting with the Pentium® processor. They are not part of the IA32 processor architecture specification. For this reason, the author has opted not to include a detailed description of these registers.

CR0 Updated

Refer to Figure 2-25 on page 90. In the 486, CR0 was updated in the following manner:

- **ET** (Extension Type) bit is hardwired. When the FPU was integrated into the later versions of the 486, CR0[ET] was hardwired to one, indicating the presence of a 387-style FPU.
- **NE** (Numeric Exception) bit added. When the FPU was integrated into the later versions of the 486, CR0[NE] (Numeric Exception) was implemented, permitting the OS to select the manner in which FP instruction exceptions are reported. See “FPU Added On-Die” on page 432 in the book for a detailed description of the NE bit.
- **WP** (Write Protect) bit added. This bit permits the OS to specify whether or not OS kernel software is permitted to write to pages defined as write-protected by their respective Page Table entries. See “The Write Protect Feature” on page 450 in the book for a detailed description of the WP bit.
- **AM** (Alignment Mask) bit added. It is used to enable or disable alignment checking on memory accesses. See “Alignment Checking Feature” on page 448 in the book for a detailed description of Alignment Checking.
- **CD** (Cache Disable) **and** **NW** (Not Write-through) bits added. They are used to control the internal cache. See “Caching-Related Changes to the Programming Environment” on page 454 in the book for a detailed description of how to control the processor’s internal cache.

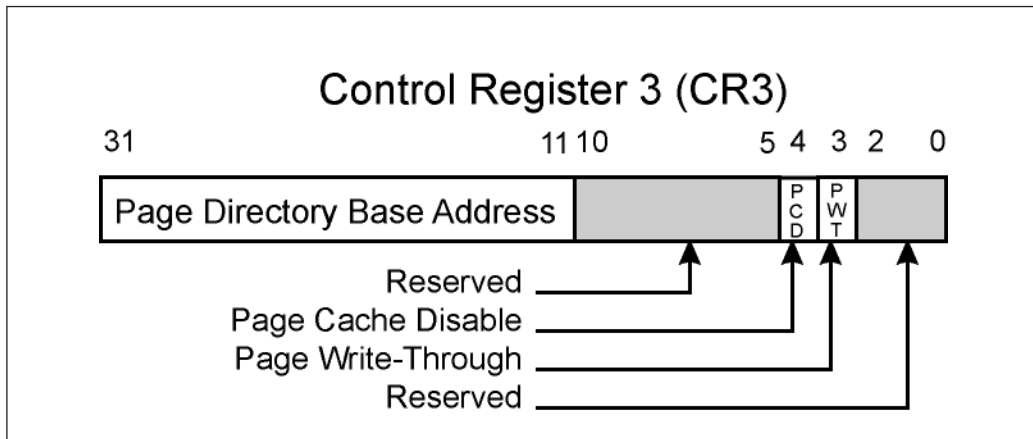
Figure 2-25: CR0 in the 486



CR3 Updated

Refer to Figure 2-26 on page 91. CR3 added the **PCD** (Page Cache Disable) and **PWT** (Page Write-Through) bits to control whether or not the processor is permitted to cache entries from the Page Directory data structure in memory and, if caching is permitted, whether to treat that page of memory as cacheable write-through or write-back memory. See “386 Demand Mode Paging” on page 209 in the book for a detailed description of Paging.

Figure 2-26: CR3 in the 486

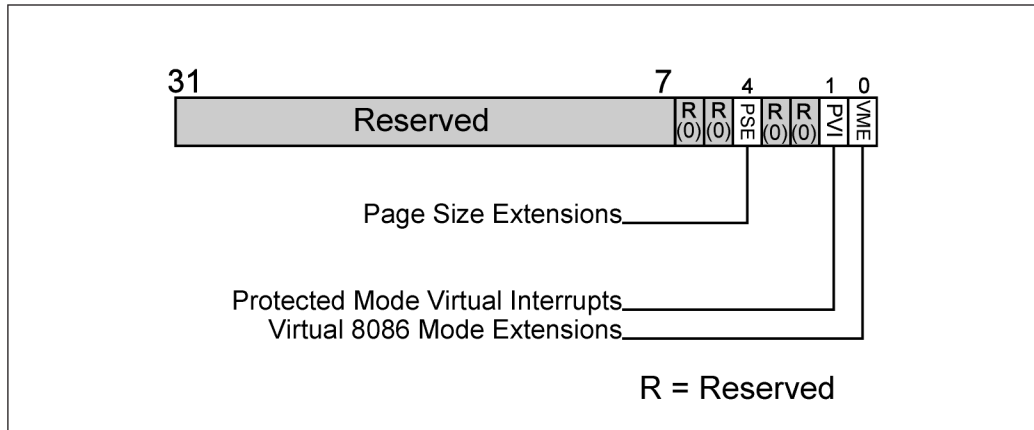


CR4 Added

Refer to Figure 2-27 on page 92. The later versions of the 486 added CR4. CR4 was first implemented in the Pentium® and was then implemented in the later versions of the 486, but only the following of the Pentium®’s CR4 bits were implemented in the 486:

- **VME (Virtual 8086 Mode Extensions) bit.** A complete description of the VME bit can be found in “Virtual 8086 Mode” on page 329 in the book.
- **PVI (Protected Mode Virtual Interrupt) bit.** A complete description of the PVI bit can be found in “Protected Mode Virtual Interrupts” on page 497 in the book.
- **PSE (Page Size Extension) bit.** A complete description of the PSE bit can be found in “4MB Pages” on page 501 in the book.

Figure 2-27: CR4 in the 486



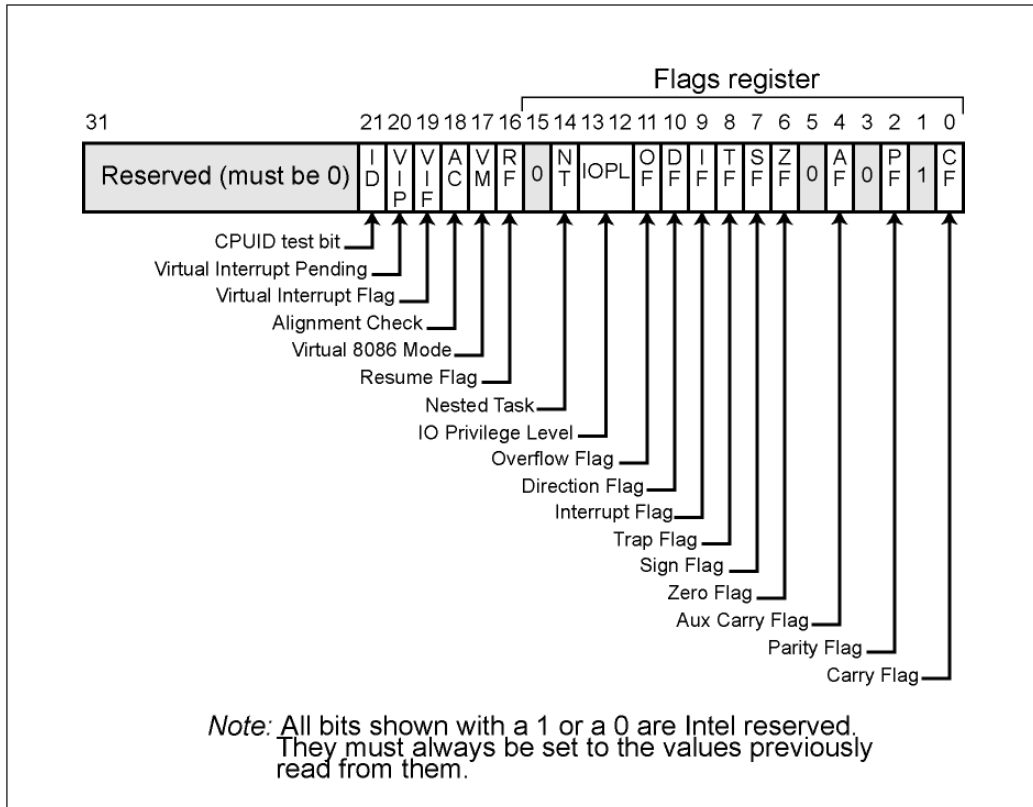
EFlags Register Updated

Refer to Figure 2-28 on page 93. The EFlags registers added the following bits:

- **AC** (Alignment Check) bit. All versions of the 486 implement this bit. See “Alignment Checking Feature” on page 448 in the book for a detailed description of Alignment Checking.
- The following EFlags bits were first implemented in the Pentium® and were then implemented in the later versions of the 486:
 - **VIF** (Virtual Interrupt Flag) bit. A complete description of the VIF bit can be found in “Efficient CLI/STI Instruction Handling” on page 492 in the book.
 - **VIP** (Virtual Interrupt Pending) bit. A complete description of the VIP bit can be found in “Efficient CLI/STI Instruction Handling” on page 492 in the book.
 - **ID (CPU ID instruction) supported bit**. A complete description of the ID bit can be found in “Determining if the CPUID instruction Is Supported” on page 1444 in the book.

Chapter 2: Register Set Evolution

Figure 2-28: EFlags Register



The Pentium® Register Set

Figure 2-29 on page 95 illustrates the Pentium®'s register set. Register set changes are briefly described in the subsections that follow and are indicated in the illustration in the following manner:

- Registers that were present in the 386/486 wherein one or more new bits were added are indicated via an asterisk (*).
- The registers that were added in the Pentium® are:
 - CR4.
 - The Local APIC register set was added in the P54C version of the Pentium®.

The Unabridged Pentium® 4

- The Machine Check Architecture registers first appeared in the Pentium®. They are implemented as Model Specific Registers (MSRs).
- The MMX register set (and instruction set) first appeared in the P55C version of the Pentium®.
- The Pentium® was the first IA32 processor to implement MSRs.
- The Pentium® was the first processor to implement a Performance Monitor register set. They were implemented as MSRs.
- The Pentium® was the first processor to implement a Time Stamp Counter register. It was implemented as an MSR.

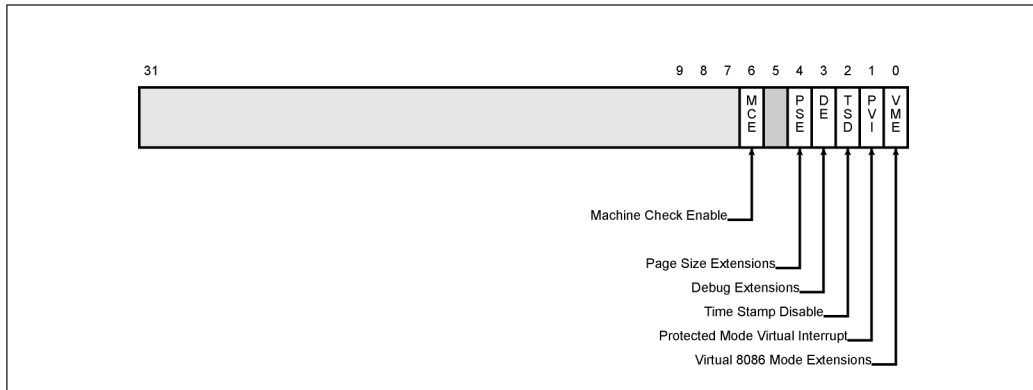
CR4 Added

Refer to Figure 2-30 on page 97. The following new bits were added in CR4:

- **VME (Virtual 8086 Mode Extensions).** When cleared to zero, VM86 mode operation is backward compatible with VM86 mode operation in the 386 (as described in “Virtual 8086 Mode” on page 329 in the book). When set to one, VM86 mode is enhanced in several ways. A detailed description of the Pentium®’s VM86 mode extension can be found in “VM86 Extensions” on page 490 in the book. CR4[VME] was also migrated into the later versions of the 486.
- **PVI (Protected Mode Virtual Interrupts).** When set to one, the PVI capability is enabled. See “Protected Mode Virtual Interrupts” on page 497 in the book for a detailed description of PVI operation. CR4[PVI] was also migrated into the later versions of the 486.
- **TSD (Time Stamp Disable).** Starting with the Pentium®, all IA32 processors incorporate a 64-bit Time Stamp Counter (TSC) that is cleared to 0 at reset and increments once for each processor clock cycle for the entire power up session (or until RESET is asserted to the processor). The RDTSC instruction is used to obtain (i.e., read) the current contents of the TSC. When CR4[TSD] is set to one, only privilege level 0 code (i.e., the OS kernel) can successfully execute the RDTSC instruction. An exception is generated if a less privileged program attempts to execute the RDTSC instruction.
- **DE (Debug Extension).** On the 386 and the 486, the Debug registers could only be used to set up memory access breakpoint recognition. If CR4[DE] = 1, the programmer can also set up IO access breakpoint recognition specifying that a Debug Exception should occur when an IO access is made to a specified IO address.
- **PSE (Page Size Extension).** On the 386 and the early models of the 486, Paging only deals with 4KB memory pages. When CR4[PSE] = 1, the OS can utilize one or more Page Directory entries to point to 4MB pages rather than to Page Tables describing 4KB pages. For a detailed description of PSE, refer to “4MB Pages” on page 501 in the book. CR4[PSE] was also migrated into the later versions of the 486.
- **MCE (Machine Check Exception Enable).** The Machine Check Architecture (MCA) was first introduced in the Pentium® in a very rudimentary form. It included a new exception (the Machine Check Exception) and the Machine Check Architecture register set. The MCA registers are actually a subset of the Model Specific Registers (MSRs). For a detailed description of MCE, refer to “MCA Enhanced” on page 588 in the book.

Chapter 2: Register Set Evolution

Figure 2-30: CR4 in the Pentium®



MMX Register Set Added

The P55C version of the Pentium® processor added MMX capability. This included a set of eight MMX data registers, MM0 through MM7, each 64 bits in width, as well as the 47 instructions that comprise the MMX instruction set. With the single exception of the Pentium® Pro processor, all subsequent IA32 processors incorporated the MMX instruction set. A detailed description of MMX can be found in “MMX Capability” on page 519 in the book.

It should be noted that the MMX registers (MM[7:0]) and the FPU data register stack, ST[7:0], are one and the same. They act as the FP data registers when executing FP instructions and as the MMX registers when executing MMX instructions. The 64-bit MMX registers are mapped into the Significand portion (i.e., the lower 64 bits) of the FP data registers.

Local APIC Added

The P54C version of the Pentium® processor was the first IA32 processor to incorporate the Local APIC (Advanced Programmable Interrupt Controller). All subsequent IA32 processors incorporated the Local APIC.

The APIC is controlled via its memory-mapped IO register set which is mapped into a 4KB block of memory space. A detailed description of the Local APIC can be found in “The Local and IO APICs” on page 1497 in the book.

Model Specific Registers (MSRs) Added

For a detailed description of the Pentium® MSRs, refer to “MSRs Added” on page 512 in the book. The MSRs are available primarily to privilege level 0 code and control items such as:

- Debug extensions.
- Performance Monitoring Counters.
- Machine Check Architecture.
- Memory Type Range Registers (MTRRs; note that they were introduced in the Pentium® Pro and are not present in the Pentium®).

Refer to Table 2-3 on page 98.

Table 2-3: Pentium® MSRs

Address	Name	Description
00h	P5_MC_ADDR	Machine Check Address register. When a FSB transaction error was detected, the address used in the transaction was latched into this register. This register is not implemented in later IA32 processors.
01h	P5_MC_TYPE	Machine Check Type register. When a FSB transaction error was detected, the transaction type was latched into this register. This register is not implemented in later IA32 processors.
02h	Test Register 1	Parity Reversal Register. Used to test parity of: • Code cache. • Data cache. • Code TLB. • Data TLB. • Microcode ROM. This register is not implemented in later IA32 processors.
03h	Reserved	

Chapter 2: Register Set Evolution

Table 2-3: Pentium® MSRs (Continued)

Address	Name	Description
04h	Test Register 2	Instruction Cache End Bit. Used to test code and data caches. This register is not implemented in later IA32 processors.
05h	Test Register 3	Cache Test Data. Used to test code and data caches. This register is not implemented in later IA32 processors.
06h	Test Register 4	Cache Test Tag. Used to test code and data caches. This register is not implemented in later IA32 processors.
07h	Test Register 5	Cache Test Control. Used to test code and data caches. This register is not implemented in later IA32 processors.
08h	Test Register 6	TLB Test Linear Address. Used to test TLBs. This register is not implemented in later IA32 processors.
09h	Test Register 7	TLB Test Control & Physical Address. Used to test TLBs. This register is not implemented in later IA32 processors.
0Ah	Reserved	
0Bh	Test Register 9	BTB Test Tag. Used to test the Branch Target Buffer (BTB). This register is not implemented in later IA32 processors.
0Ch	Test Register 10	BTB Test Target. Used to test the Branch Target Buffer (BTB). This register is not implemented in later IA32 processors.
0Dh	Test Register 11	BTB Test Control. Used to test the Branch Target Buffer (BTB). This register is not implemented in later IA32 processors.
0Eh	Test Register 12	New Feature Control. Refer to Figure 21-16 on page 516 in the book and “Test Register 12” on page 515 in the book. This register is not implemented in later IA32 processors.
0Fh	Reserved	

Table 2-3: Pentium® MSRs (Continued)

Address	Name	Description
10h	TSC	Time Stamp Counter. This register is implemented in all later IA32 processors.
11h	CESR	Performance Monitor Control and Event Select register. Refer to “Performance Monitoring” on page 505 in the book. The Performance Monitoring register set is implemented differently on later IA32 processors.
12h	CTR0	Performance Monitor Counter 0. Refer to “Performance Monitoring” on page 505 in the book. The Performance Monitoring register set is implemented differently on later IA32 processors.
13h	CTR1	Performance Monitor Counter 1. Refer to “Performance Monitoring” on page 505 in the book. The Performance Monitoring register set is implemented differently on later IA32 processors.
14h-FFh	Reserved	

Machine Check Architecture Registers Added

For a detailed description of MCA as implemented in the Pentium® processor, refer to “Machine Check Architecture (MCA)” on page 504 in the book.

The Pentium® processor was the first IA32 processor to implement a set of registers that are used to log hardware-related errors detected within the processor or on its FSB. Collectively, this error logging facility is referred to as the Machine Check Architecture (MCA) and has been incorporated in all subsequent IA32 processors. The MCA registers are implemented as MSRs and therefore are accessed using the RDMSR and WRMSR instructions. The Pentium® implemented a rudimentary set of MCA registers.

Performance Monitoring Registers Added

The Performance Monitoring registers are implemented as MSRs and therefore are accessed using the RDMSR and WRMSR instructions. They consists of:

Chapter 2: Register Set Evolution

- Two 40-bit event counting registers (PerfCtr0 MSR and PerfCtr1 MSR).
- The 32-bit CESR (Control and Event Select Register) MSR that is used to set up one, the other, or both counters to count the number of, or measure the duration of, different internal processor event types.

A more detailed description of the Pentium®'s performance monitoring facility can be found in "Performance Monitoring" on page 505 in the book.

Time Stamp Counter (TSC) Added

Starting with the Pentium®, all IA32 processors incorporate a 64-bit Time Stamp Counter (TSC) that is cleared to 0 at reset and increments once for each processor clock cycle for the entire power up session (or until RESET is asserted to the processor). The RDTSC instruction is used to obtain (i.e., read) the current contents of the TSC. When CR4[TSD] is set to one, only privilege level 0 code (i.e., the OS kernel) can successfully execute the RDTSC instruction. An exception is generated if a less privileged program attempts to execute the RDTSC instruction.

DR7 Enhanced

The Debug Control register (DR7) was enhanced so that, if CR4[DE] = 1, debug software can use the debug registers to set up IO access recognition as well as memory access recognition. See "The Debug Registers" on page 375 in the book for a detailed description of the Pentium®'s debug register set.

Test Registers Deleted

The Pentium® deleted TR3 through TR7 (used to test the on-chip caches and TLBs in the 386 and 486) and moved their functionality into the MSRs. See "MSRs Added" on page 512 in the book for a detailed description of the Pentium®'s MSRs.

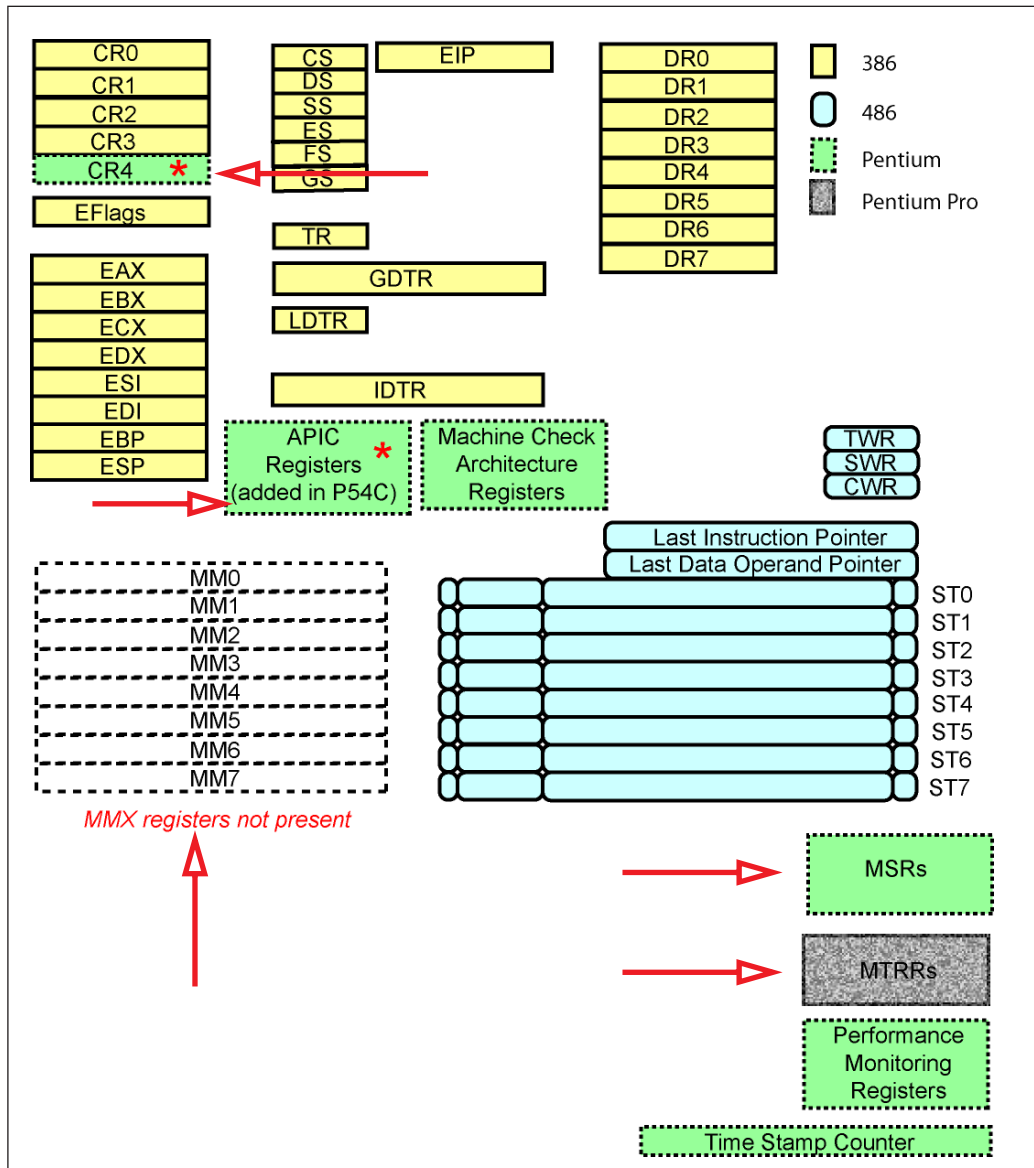
The Pentium® Pro Register Set

Figure 2-31 on page 102 illustrates the Pentium® Pro's register set. Register set changes are briefly described in the subsections that follow and are indicated in the illustration in the following manner:

The Unabridged Pentium® 4

- Registers that were present in the 386, 486 and Pentium® wherein one or more new bits were added are indicated via an asterisk (*).
- The new registers that were added in the Pentium® Pro consists of MSRs and the Local APIC registers.

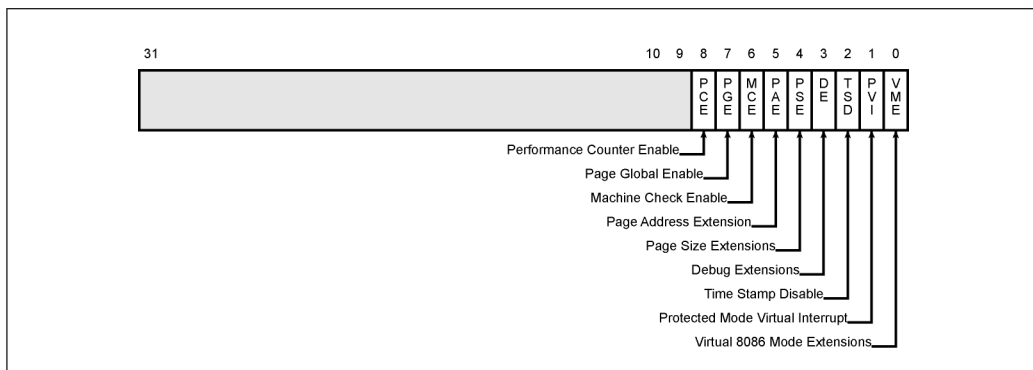
Figure 2-31: Pentium® Pro Register Set



Refer to Figure 2-32 on page 103. CR4 added the following new feature control bits:

- **Physical Address Extension (PAE).** Refer to “PAE-36 Mode” on page 554 in the book for a detailed description of the PAE feature:
 - 0 = The PAE feature is disabled and the processor can only address the lower 4GB of memory address space.
 - 1 = The PAE feature is enabled and the processor can address 2^{36} (64GB) of memory space.
- **Page Global Enable (PGE).** Refer to “Global Pages” on page 567 in the book for a detailed description of the PGE feature:
 - 0 = The PGE feature is disabled and, whenever CR3 (the Page Directory Base Address Register—PDBR) is written to, all Page Table Entries (PTEs) cached in the processor’s TLB(s) are deleted.
 - 1 = The PGE feature is enabled and, whenever CR3 is written to, all Page Table Entries cached in the processor’s TLB(s) are deleted with the exception of those with the G bit set to one.
- **RDPMC instruction Enable (PCE).** Refer to “The Performance Counters” on page 606 in the book for a detailed description of the Pentium® Pro’s Performance Monitoring facility:
 - 0 = Only privilege level 0 code (i.e., the OS kernel) can execute the RDPMC instruction to read a performance counter’s contents. An exception is generated whenever a less-privileged program attempts execution of the RDPMC instruction.
 - 1 = Any program can execute the RDPMC instruction to read a performance counter’s contents.

Figure 2-32: CR4 in the Pentium® Pro



Additions to the Local APIC Register Set

Added Performance Counter Overflow Interrupt

The Pentium® Pro added an entry to the Local APIC's LVT (Local Vector Table) to handle an interrupt on Performance Counter overflow. See "The Performance Counter Overflow Interrupt" on page 1547 in the book and "Enabling PEBS" on page 1415 in the book for a detailed description of the Performance Counter overflow interrupt.

APIC's Base Address Assignable

While the base memory address for the Pentium®'s Local APIC register set was fixed at address FEE00000h, an MSR was added that permits the programmer to assign a different base address. See "Local Cluster and APIC ID Assignment" on page 1513 in the book for a detailed description of the APIC_BASE MSR.

MMX Not Implemented

With the exception of the Pentium® Pro, the MMX register set and instruction set were implemented in all IA32 processors starting with the P55C version of the Pentium® processor.

New MSRs

The Pentium® Pro greatly expanded the number of MSRs implemented in the IA32 processors.

The following new MSRs were implemented:

- APIC_BASE MSR.
- EBL_CR_POWERON MSR.
- TEST_CTL.
- DEBUGCTLMSR.
- ROB_CR_BKUPTMPDR6.
- Microcode Update MSRs.
- Performance Monitoring MSRs.
- Memory Type and Range Registers (MTRRs).
- MCA Registers.
- Branch Recording Registers.

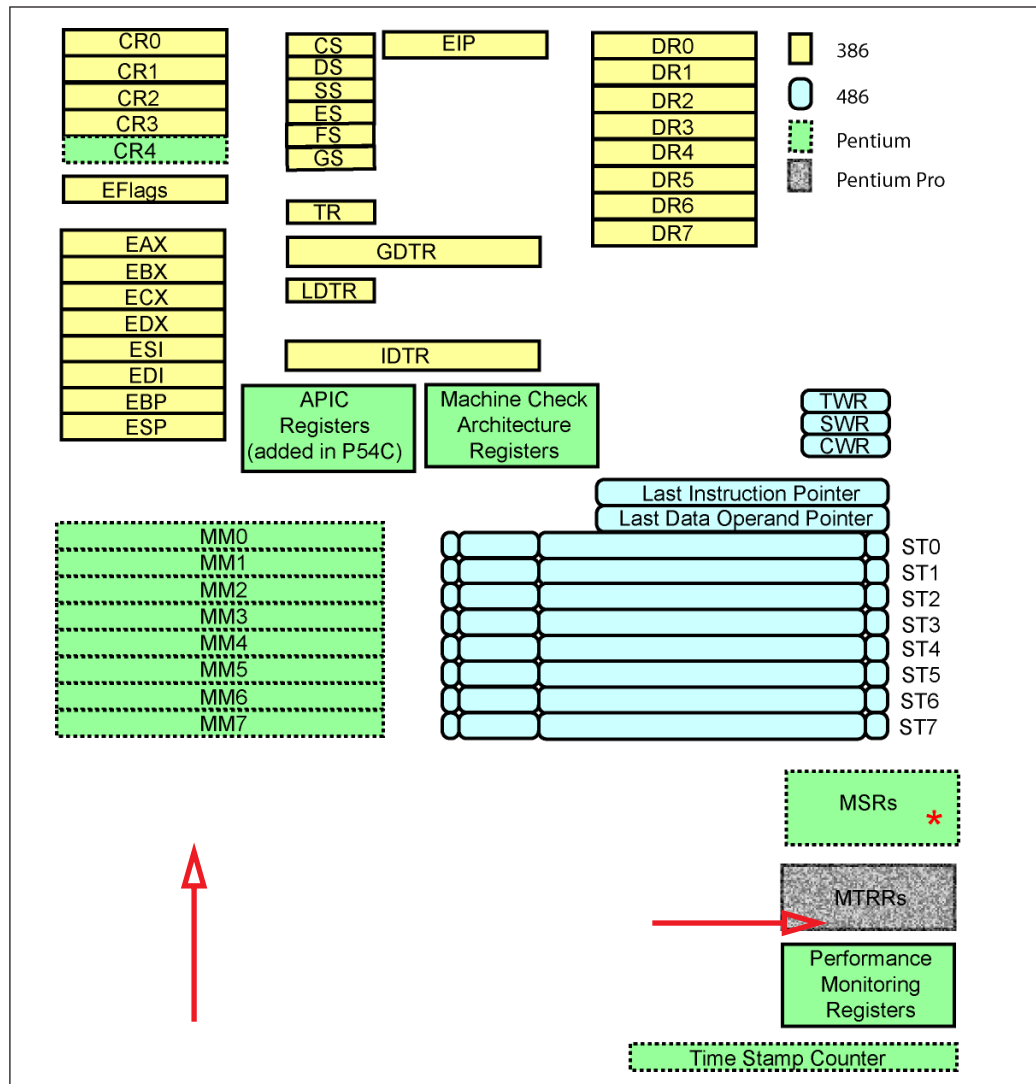
Refer to "MSRs Added" on page 612 in the book for a detailed description of the Pentium® Pro's MSRs.

The Pentium® II Register Set

Figure 2-33 on page 106 illustrates the Pentium® II's register set. Unlike the Pentium® Pro, the Pentium® II did implement the MMX register set and instruction set. The only other register set changes between the Pentium® Pro and the Pentium® II were in the MSRs.

The Unabridged Pentium® 4

Figure 2-33: Pentium® II Register Set



MMX Added Back In

Although not present in the Pentium® Pro, the MMX register set and instruction set were implemented in the Pentium® II and all subsequent IA32 processors. See “MMX Capability” on page 519 in the book for a detailed description of MMX.

Pentium® II and III MSR Changes

The following new MSRs were implemented:

- IA32_Platform_ID MSR.
- Fast System Enter / Exit Registers.
- IA32_CR_PAT.
- BBL (Backside Bus Logic) registers.

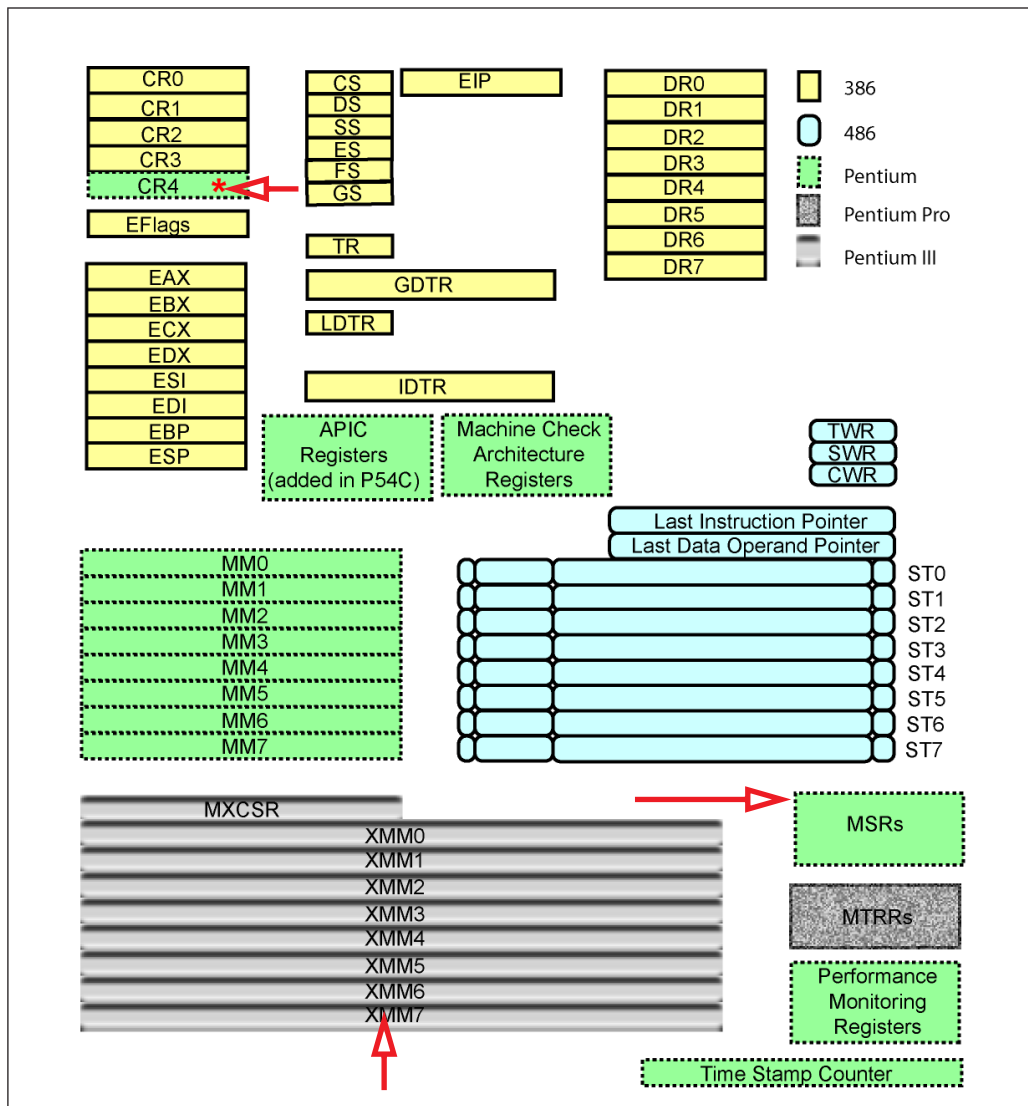
Refer to “The Pentium® II and Pentium® III MSRs” on page 696 in the book for a detailed description of the Pentium® II’s and the Pentium® III’s MSRs.

The Pentium® III Register Set

Figure 2-34 on page 108 illustrates the Pentium® III’s register set. The Pentium® III added two new feature control bits in CR4 and also added the SSE (Streaming SIMD Extensions) registers and instruction set. With one exception, it has the same MSRs as the Pentium® II. The exception was the addition of the IA32_PLATFORM_ID MSR register.

The Unabridged Pentium® 4

Figure 2-34: Pentium® III Register Set



Chapter 2: Register Set Evolution

New CR4 Bits

Table 2-4 on page 109 lists the new feature control bits in CR4 added by the Pentium® III.

Table 2-4: Pentium® III Added New Bits in CR4

Bit Name	Description
OSFXSR	Set by the OS if it supports the SSE instruction set. Refer to “The Streaming SIMD Extensions (SSE)” on page 758 in the book for a detailed description of the SSE register set and instruction set: • 0: The FXSAVE and FXRSTOR instructions save and restore the contents of the FPU and MMX registers, but they don’t save and restore the contents of the XMM and MXCSR registers. • 1: Has the following effects: - Indicates to application software that the OS supports execution of the FXSAVE and FXRSTOR instructions. - Enables the FXSAVE and FXRSTOR instructions to save and restore the XMM and MXCSR registers in addition to the FPU and MMX registers. - Enables the processor to execute the SSE instructions, with the exception of PREFETCHH and SFENCE (which are always enabled).
OSXMMEXCPT	Set by the OS if it supports the SSE FP Exception. Refer to “The Streaming SIMD Extensions (SSE)” on page 758 in the book for a detailed description of the SSE register set and instruction set: • 1: Indicates that the OS’s SSE FP exception handler supports the handling of unmasked SIMD FP exceptions. SIMD FP exceptions are only generated by SSE SIMD FP instructions. • 0: The processor generates an invalid opcode exception whenever it detects an unmasked SIMD FP exception.

SSE Register Set Added

The Pentium® III added the 70 new instructions that comprise the SSE instruction set (originally referred to as the Katmai New Instructions, or KNI). The SSE register set was also added and is comprised of:

- Eight XMM data registers (XMM[7:0]), each 128-bits in width.
- The 32-bit MXCSR (SSE Control and Status Register).

Refer to “The Streaming SIMD Extensions (SSE)” on page 758 in the book for a detailed description of the SSE register set and instruction set.

IA32_PLATFORM_ID MSR Added

The IA32_PLATFORM_ID MSR was added to enhance the Microcode Update feature. For more information, refer to “MicroCode Update Feature” on page 631 in the book.

The Pentium® 4 and Pentium® M Register Set

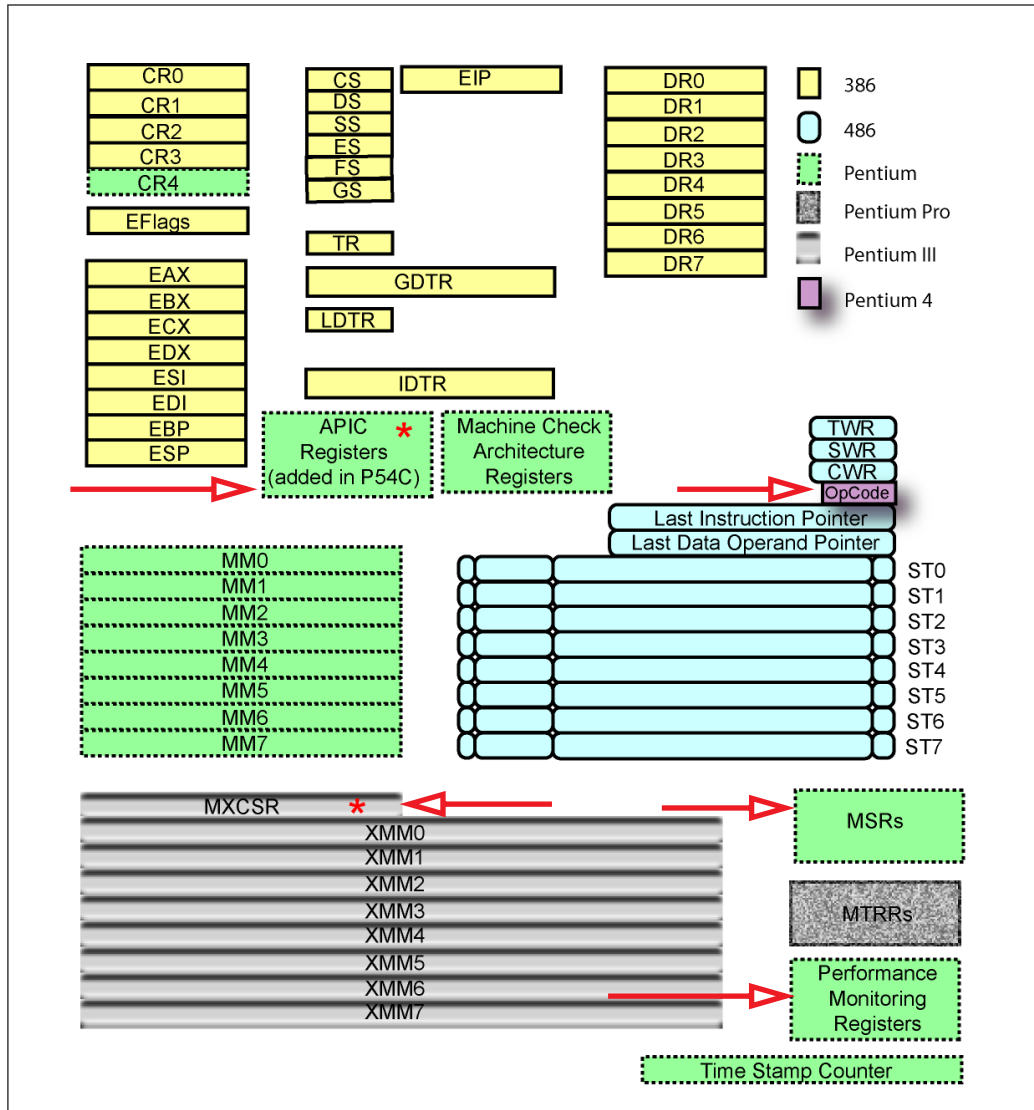
Figure 2-35 on page 111 illustrates the register set of the Pentium® 4 and the Pentium® M. The register set changes consists of the following:

- Some changes to the Local APIC.
- The 3-wire APIC bus was deleted (not a register set change, but a change nonetheless).
- A new interrupt message transaction was added on the FSB.
- Addition of the FP OpCode register.
- Addition of one control bit to the MXCSR register.
- Changes to the MSR register set.
- Expanded the number of Performance Counters.

The sections that follow provide a basic description of these changes.

Chapter 2: Register Set Evolution

Figure 2-35: Pentium® 4 and Pentium® M Register Set



FP Opcode Register Added

In the earlier IA32 processors, the following information was automatically captured when a FP exception was detected:

- The memory address of the offending instruction was recorded in the Last Instruction Pointer register and,
- If the instruction referenced a data operand in memory, the address of the data operand was also recorded in the Last Data Operand Pointer register.

The Pentium® 4 processor added the FP Opcode register. When the execution of a non-control FP instruction (i.e., a calculation) causes a FP exception, the opcode of the offending instruction is automatically recorded in the Opcode register. Rather than having to use the address stored in the Last Instruction Pointer register to obtain the opcode of the FP instruction that caused the exception, a FP exception handler written for the Pentium® 4 or Pentium® M can obtain the opcode directly from the Opcode register. A complete description of this register can be found in “FPU Enhancement” on page 1345 in the book.

DAZ Bit Added to MXCSR

A new control bit, Denormals Are Zeros (DAZ), was added to the MXCSR register. For more information, refer to the detailed description of the SSE2 instruction set in “The SSE2 Instruction Set” on page 1332 in the book.

Local APIC Changes

The following changes were made to the Local APIC:

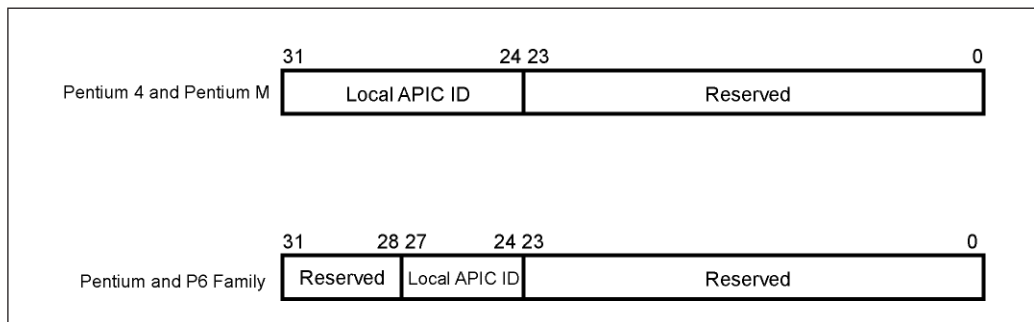
- Refer to Figure 2-36 on page 113. The Thermal Sensor interrupt was added and a corresponding entry was added to the Local Vector Table (LVT) in the Local APIC’s register set.
- The APIC bus Arbitration ID register was deleted because the APIC bus has been eliminated. Interrupts are now delivered to the processor via a memory write transaction on the FSB.
- Refer to Figure 2-37 on page 113. The format of the APIC ID register has changed. For the P6 and Pentium®, the APIC ID field is 4 bits in width, and encodings 0h through Eh can be used to uniquely identify one of 15 different processors connected to the APIC bus. For the Pentium® 4 and Pentium® M, the xAPIC spec extends the local APIC ID field to 8 bits which can be used to identify one of 255 processors in the system.

Chapter 2: Register Set Evolution

Figure 2-36: Thermal Sensor Interrupt Entry Added to the LVT

		FEE0 01E0h
		FEE0 01F0h
Reserved	Interrupt Request Register (IRR) 256 bits	FEE0 0200h
		FEE0 0210h
		FEE0 0220h
		FEE0 0230h
		FEE0 0240h
		FEE0 0250h
		FEE0 0260h
		FEE0 0270h
Reserved	Error Status Register	FEE0 0280h
Reserved		FEE0 0290h
		FEE0 02A0h
		FEE0 02B0h
		FEE0 02C0h
		FEE0 02D0h
		FEE0 02E0h
		FEE0 02F0h
Reserved	Interrupt Command Register[31:0]	FEE0 0300h
Reserved	Interrupt Command Register[63:32]	FEE0 0310h
Reserved	LVT—Timer	FEE0 0320h
Reserved	LVT—Thermal Sensor	FEE0 0330h
Reserved	LVT—Perf Counter OverFlow	FEE0 0340h
Reserved	LVT—LINT0	FEE0 0350h
Reserved	LVT—LINT1	FEE0 0360h
Reserved	LVT—Error	FEE0 0370h
Reserved	Timer Initial Count Register	FEE0 0380h
Reserved	Timer Current Count Register	FEE0 0390h
Reserved		FEE0 03A0h
		FEE0 03B0h
		FEE0 03C0h
		FEE0 03D0h
	Timer Divide Config Register	FEE0 03E0h
Reserved		FEE0 03F0h

Figure 2-37: APIC ID Register Format Changed



MSR Register Set Changes

The following new MSRs were implemented:

- A number of the MSRs are designated as part of the IA32 processor architecture and as such are guaranteed to remain at the same MSR addresses with the same format in future members of the IA32 processor family.
- MSR_EBC_HARD_POWERON.
- MSR_EBC_SOFT_POWERON.
- MSR_EBC_FREQUENCY_ID.
- IA32_MISC_ENABLE.
- IA32_DS_AREA.
- The number of performance counters has been increased from 2 to 18.
- Improved Branch Recording Registers.
- Thermal Monitor related Registers.

Refer to “The MSRs” on page 1347 in the book for a detailed description of the Pentium® 4’s MSRs.

3

Instruction Set Evolution

Introduction

The processor's only purpose in life is to fetch, decode, and execute programs written by programmers. The processor automatically fetches the instructions that comprise a program from memory, decodes them and executes them. Starting with the 386 and proceeding to today's Pentium® 4 and Pentium® M processors, this chapter provides an overview of the evolution of the IA32 processor instruction set.

386 Instruction Set

Table 3-1 on page 116 lists the 386 processor's instruction set.

The Unabridged Pentium® 4

Table 3-1: 386 Instruction Set

Instruction	Description
Data Transfer—General Purpose	
MOV	Move operand
PUSH	Push operand onto stack
POP	Pop operand off stack
PUSHA	Push all registers on stack
POPA	Pop all registers off stack
XCHG	Exchange Operand, Register
XLAT	Translate
Data Transfer—Conversion	
MOVZX	Move byte or Word, Dword, with zero extension
MOVSX	Move byte or Word, Dword, sign extended
CBW	Convert byte to Word, or Word to Dword
CWD	Convert Word to DWORD
CWDE	Convert Word to DWORD extended
CDQ	Convert DWORD to QWORD
Data Transfer—Input/Output	
IN	Input operand from I/O space
OUT	Output operand to I/O space
Data Transfer—Address Object	
LEA	Load effective address
LDS	Load pointer into D segment register
LES	Load pointer into E segment register
LFS	Load pointer into F segment register

Chapter 3: Instruction Set Evolution

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
LGS	Load pointer into G segment register
LSS	Load pointer into S (Stack) segment register
Data Transfer—Flag Manipulation	
LAHF	Load A register from Flags
SAHF	Store A register in Flags
PUSHF	Push flags onto stack
POPF	Pop flags off stack
PUSHFD	Push EFlags onto stack
POPFD	Pop EFlags off stack
CLC	Clear Carry Flag
CLD	Clear Direction Flag
CMC	Complement Carry Flag
STC	Set Carry Flag
STD	Set Direction Flag
Arithmetic Instructions - Addition	
ADD	Add operands
ADC	Add with carry
INC	Increment operand by 1
AAA	ASCII adjust for addition
DAA	Decimal adjust for addition
Arithmetic Instructions - Subtraction	
SUB	Subtract operands
SBB	Subtract with borrow

The Unabridged Pentium® 4

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
DEC	Decrement operand by 1
NEG	Negate operand
CMP	Compare operands
DAS	Decimal adjust for subtraction
AAS	ASCII Adjust for subtraction
Arithmetic Instructions - Multiplication	
MUL	Multiply Double/Single Precision
IMUL	Integer multiply
AAM	ASCII adjust after multiply
Arithmetic Instructions - Division	
DIV	Divide unsigned
IDIV	Integer Divide
AAD	ASCII adjust before division
String Instructions	
MOVS	Move byte or Word, Dword string
INS	Input string from I/O space
OUTS	Output string to I/O space
CMPBS	Compare byte or Word, Dword string
SCAS	Scan Byte or Word, Dword string
LODS	Load byte or Word, Dword string
STOS	Store byte or Word, Dword string
REP	Repeat
REPE/REPZ	Repeat while equal/zero

Chapter 3: Instruction Set Evolution

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
RENE/REPNZ	Repeat while not equal/ not zero
Logical Instructions-Logicals	
NOT	NOT' operands
AND	AND' operands
OR	Inclusive OR operands
XOR	Exclusive OR operands
TEST	Test operands
Logical Instructions-Shifts	
SHL/SHR	Shift logical left or right
SAL/SAR	Shift arithmetic left or right
SHLD/SHRD	Double shift left or right
Logical Instructions-Rotates	
ROL/ROR	Rotate left/right
RCL/RCR	Rotate through carry left/right
Bit Manipulation Instructions	
BT	Bit Test
BTS	Bit Test and Set
BTR	Bit Test and Reset
BTC	Bit Test and Complement
BSF	Bit Scan Forward
BSR	Bit Scan Reverse

The Unabridged Pentium® 4

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
Program Control Instructions-Conditional Transfers	
SETCC	Set byte equal to condition code
JA/JNBE	Jump if above/not below nor equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above nor equal
JBE/JNA	Jump if below or equal/not above
JC	Jump if carry
JE/JZ	JE/JZ Jump if equal/zero
JG/JNLE	Jump if greater/not less nor equal
JGE/JNL	Jump if greater or equal/not less
JL/JNGE	Jump if less/not greater nor equal
JLE/JNG	Jump if less or equal/not greater
JNC	Jump if not carry
JNE/JNZ	Jump if not equal/not zero
JNO	Jump if not overflow
JNP/JPO	Jump if not parity/parity odd
JNS	Jump if not sign
JO	Jump if overflow
JP/JPE	Jump if parity/parity even
JS	Jump if Sign
Program Control Instructions-Unconditional Transfers	
CALL	Call procedure/task
RET	Return from procedure

Chapter 3: Instruction Set Evolution

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
JMP	Jump
Program Control Instructions-Iteration Controls	
LOOP	Loop
LOOPE/LOOPZ	Loop if equal/zero
LOOPNE/LOOPNZ	Loop if not equal/not zero
JCXZ	JUMP if register CXe0
Program Control Instructions-Interrupts	
INT	Interrupt
INTO	Interrupt if overflow
IRET	Return from Interrupt/Task
CLI	Clear Interrupt Enable
STI	Set Interrupt Enable
High Level Language Instructions	
BOUND	Check Array Bounds
ENTER	Setup Parameter Block for Entering Procedure
LEAVE	Leave Procedure
Protection Model	
SGDT	Store Global Descriptor Table
SIDT	Store Interrupt Descriptor Table
STR	Store Task Register
SLDT	Store Local Descriptor Table
LGDT	Load Global Descriptor Table
LIDT	Load Interrupt Descriptor Table

The Unabridged Pentium® 4

Table 3-1: 386 Instruction Set (Continued)

Instruction	Description
LTR	Load Task Register
LLDT	Load Local Descriptor Table
ARPL	Adjust Requested Privilege Level
LAR	Load Access Rights
LSL	Load Segment Limit
VERR/VERW	Verify Segment for Reading or Writing
LMSW	Load Machine Status Word (lower 16 bits of CR0)
SMSW	Store Machine Status Word
Processor Control Instructions	
HLT	Halt
WAIT	Wait until BUSY negated
ESC	Escape
LOCK	Lock Bus

486 Additions to the Instruction Set

Refer to Table 3-2 on page 123. The 486 added seven new instructions to the 386 instruction repertoire.

Note that the CUID instruction was first introduced in the Pentium® and then migrated into the later generations of the 486. It became a standard part of the instruction set from that point forward.

Chapter 3: Instruction Set Evolution

Table 3-2: 486 Instruction Set Additions

Instruction	Description
XADD	Exchange and Add. Can be used to optimize parallel loop execution. Exchanges the destination operand with the source operand, then loads the sum of the two values into the destination operand. Destination operand can be a register or a memory location; the source operand is a register. When used with the LOCK prefix, it's executed as a locked RMW operation.
CMPXCHG	Compare and Exchange. Intended for use in testing and manipulating software semaphores. Compares the value in AL, AX, or EAX with the destination operand. If the two values are equal, the source operand is loaded into the destination operand. Otherwise, the destination operand is loaded into AL, AX, or EAX. When used with a LOCK prefix, it's executed as a locked RMW operation. The destination operand receives a write without regard to the comparison's result (the processor never performs a locked read without subsequently performing a locked write). The destination operand is written back if the comparison fails; otherwise, the source operand is written into the destination.
INVD	Invalidate Cache. All lines in the cache are invalidated (including modified lines in the writeback version of the 486). In addition, the processor executes the Flush version of the Special transaction on the FSB to command an external cache (if one is present) to invalidate its contents as well (including any modified lines).
WBINVD	Write Back and Invalidate. This instruction only appeared in the Write Back Enhanced version of the 486 (as well as all subsequent IA32 processors). All other versions of the 486 had Write Through rather than Write Back caches and therefore did not implement this instruction.
INVLPG	Invalidate TLB Entry. Invalidates the Translation Lookaside Buffer (TLB) entry specified with the source operand (a memory address). The processor determines the page that contains that address and flushes the TLB entry for that page).

The Unabridged Pentium® 4

Table 3-2: 486 Instruction Set Additions (Continued)

Instruction	Description
RSM	Resume from System Management Mode. The Resume instruction actually first appeared in the 386SL variant of the 386, but did not become part of the standard instruction set until the later generations of the 486. From that point forward, SMM (and therefore the RSM instruction) became standard in all subsequent processors.
BSWAP	Byte Swap. Reverses the byte order in a 32-bit register. When data is transferred between a register and memory by an IA32 processor, the order used is always little-endian. Software written for the VAX also used the little-endian methodology. Since many processor types (e.g., PowerPC and Power) store data in the reverse order (big-endian), BSWAP can be used to reorder data to allow efficient access to existing databases built by big-endian machines.

Pentium® Additions to the Instruction Set

Refer to Table 3-3 on page 125. The initial versions of the Pentium® added five new instructions to the 486 instruction repertoire. In addition, starting with the P55C version of the Pentium®, the Pentium® added the 47 instructions that comprise the MMX instruction set. With the single exception of the Pentium® Pro processor, all subsequent processors incorporated the MMX instruction set.

Figure Figure 3-1 on page 126 illustrates the capabilities reported in the EDX register by a CUID request type 1.

Chapter 3: Instruction Set Evolution

Table 3-3: Pentium® Instruction Set Additions/Changes

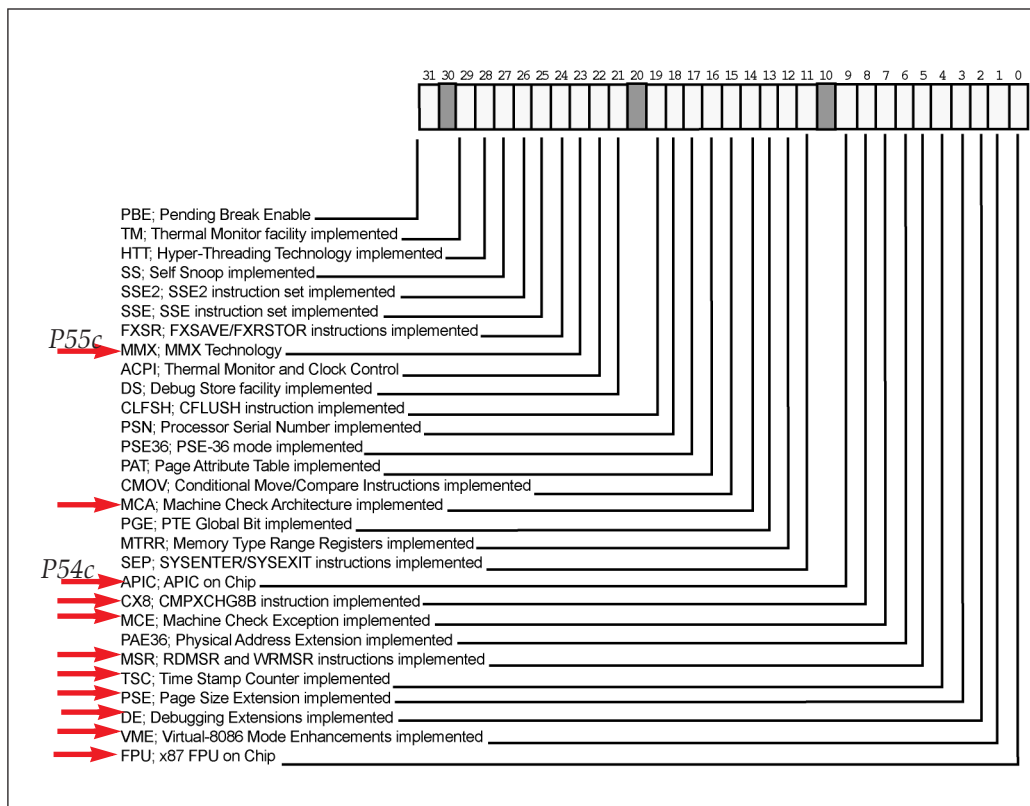
Instruction	Description
CMPXCHG8B	Compare and Exchange Eight Bytes. Compares two 8-byte data objects. Compares the 64-bit value in EDX:EAX with a value in memory: • If equal, the value in ECX:EBX is stored into the specified memory operand. • If unequal, the contents of the memory operand is copied into EDX:EAX. Can be used with a LOCK prefix to execute it as an atomic operation (the processor never produces a locked read without producing a subsequent locked write). The destination operand always receives a write regardless of the compare result. The destination operand is written back if the compare fails; otherwise, the source operand is written into the destination.
CPUID	CPU identification. Refer to “CPU Identification” on page 1443 in the book for more information.
RDTSC	Read Time Stamp Counter. Loads the current 64-bit TSC value into EDX:EAX (the upper 32 bits into EDX and the lower 32 bits into EAX). TSC is incremented every clock cycle and is reset to 0 when the processor is reset. When in Protected or VM86 mode, CR4[TSD] (Time Stamp Disable) restricts RDTSC use as follows: • TSD = 0: RDTSC can be executed at any privilege level. • TSD = 1: Can only be executed at privilege level 0. • When in Real mode, RDTSC execution is always permitted. TSC can also be read with RDMSR at privilege level 0. On a post-Pentium® processor, RDTSC is not a serializing instruction (see “Serializing Instructions” on page 1079 in the book). It does not necessarily wait until all previous instructions are executed before reading the TSC. Subsequent instructions may begin execution before the read is performed.
RDMSR	Read and Write Model Specific Register.
WRMSR	

The Unabridged Pentium® 4

Table 3-3: Pentium® Instruction Set Additions/Changes (Continued)

Instruction	Description
MMX	In addition, the P55C version of the Pentium® added the 47 instructions that comprise the MMX instruction set. With the single exception of the Pentium® Pro processor, all subsequent processors incorporated the MMX instruction set.

Figure 3-1: Pentium® Capabilities



Pentium® Pro Additions to the Instruction Set

The Pentium® Pro made the additions to the instruction set indicated in Table 3-4 on page 127. Unless otherwise noted, all of these are new instructions added to the instruction set (rather than updates to preexistent instructions).

Figure 3-2 on page 129 illustrates the capabilities reported in the EDX register by a CPUID request type 1.

Table 3-4: Pentium® Pro Instruction Set Additions/Changes

Instruction	Description
CMOV	Conditional move: • Problem it addresses: Mispredicted branches cause a deep dip in performance, so, wherever possible, the conversion of conditional branches into conditional moves is a good thing. • Description: Checks the state of one or more EFlags status bits (CF, OF, PF, SF, and ZF) and performs a move if the condition is met. If the condition is not satisfied, a move is not performed and execution continues with the instruction following the CMOV. Can move a 16- or 32-bit value from memory to a GPR or from one GPR to another. Moves of 8-bit register operands are not supported. Software can determine if CMOV is supported in a specific processor using the CPUID instruction.
FCMOV	FP conditional move: • Problem addressed: Mispredicted branches cause a deep dip in performance, so, wherever possible, the conversion of conditional branches into conditional moves is a good thing. • New Instructions. The new FP-related instructions are: - FCOMI (FP compare and set EFlags). - FCMOV (FP conditional move). FCOMI is executed first to perform the compare and set the EFlags status bits appropriately, after which the conditional FP move (FCMOV) is executed. The value lies in the fact that it eliminates a conditional branch instruction and therefore the chance of a branch misprediction (and the subsequently sharp dip in performance).

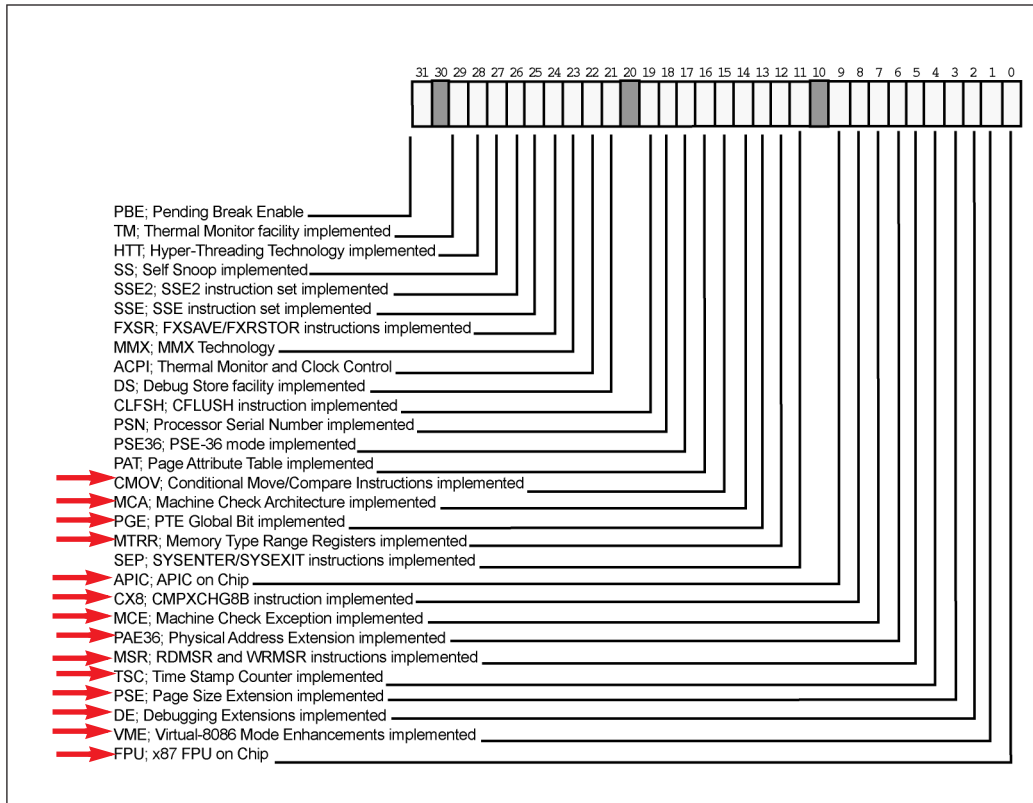
The Unabridged Pentium® 4

Table 3-4: Pentium® Pro Instruction Set Additions/Changes (Continued)

Instruction	Description
FCOMI	FP compare and set EFlags. See the previous row.
RDPMC	Read performance monitoring counters. • Problem addressed: Although the Performance Monitoring Counters were implemented in the pre-MMX Pentium®, they could only be accessed by privilege level 0 code using the RDMSR and WRMSR instructions. This meant that code running at a lesser privilege level (1, 2, or 3) could not utilize the Performance Monitoring facility. RDPMC loads the contents of the specified (in ECX) 40-bit counter into EDX:EAX. EDX is loaded with the upper 8 bits and EAX with the lower 32 bits. A counter (0 or 1) is specified by placing 0000h or 0001h, respectively, in ECX. Allows code running at a lower privilege level to read the counters if CR4[PCE] = 1. Allows performance monitoring by code without incurring the overhead of an OS call. RDPMC isn't a serializing instruction (see "Serializing Instructions" on page 1079 in the book). Events caused by earlier instructions may not have completed, or events caused by subsequent instructions may not have begun. If an exact event count is required, use a serializing instruction (such as CPUID) before and/or after the RDPMC instruction. RDPMC can execute in 16-bit mode or VM86 mode, but the full contents of ECX is used to identify the counter to access and the 40-bit result is returned in EDX:EAX.
UD2	Architecturally defined "undefined" instruction. Provided for software testing to explicitly generate an invalid opcode exception. Its opcode is reserved for this purpose. Other than raising the invalid opcode exception, it is the same as a NOP.
CPUID	Enhanced to return additional information. Refer to "Enhanced Processor Signature" on page 1460 in the book for additional information.

Chapter 3: Instruction Set Evolution

Figure 3-2: Pentium® Pro Capabilities



Pentium® II Additions to the Instruction Set

Table 3-5 on page 130 identifies the new instructions added in the Pentium® II.

Figure 3-3 on page 131 illustrates the capabilities reported in the EDX register by a CPUID request type 1.

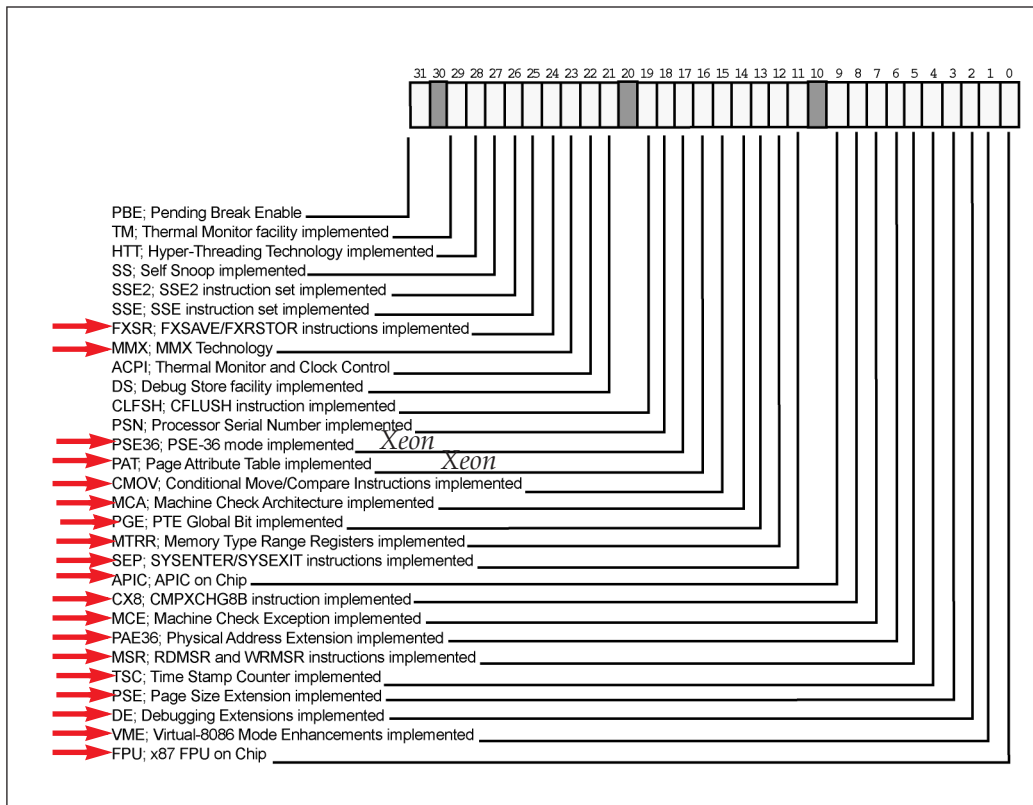
The Unabridged Pentium® 4

Table 3-5: Pentium® II Instruction Set Additions/Changes

Instruction	Description
FXRSTOR	Floating Point or MMX Restore. Restore FP or MMX State. See “FP/SSE Save/Restore Instruction Pair” on page 712 in the book for more information.
FXSAVE	Floating Point or MMX Save. Store FP or MMX State. See “FP/SSE Save/Restore Instruction Pair” on page 712 in the book for more information.
SYSENTER	Kernel System procedure Enter. Fast Transition to the System Call Entry Point. See “Fast System Call/Return Instruction Pair” on page 708 in the book for more information.
SYSEXIT	Kernel System procedure Exit. Fast Transition from the System Call Entry Point. See “Fast System Call/Return Instruction Pair” on page 708 in the book for more information.

Chapter 3: Instruction Set Evolution

Figure 3-3: Pentium® II Capabilities



Pentium® III Additions to the Instruction Set

Table 3-6 on page 132 lists the new instructions that the Pentium® III added to the instruction set.

Figure 3-4 on page 133 illustrates the capabilities reported in the EDX register by a CPUID request type 1.

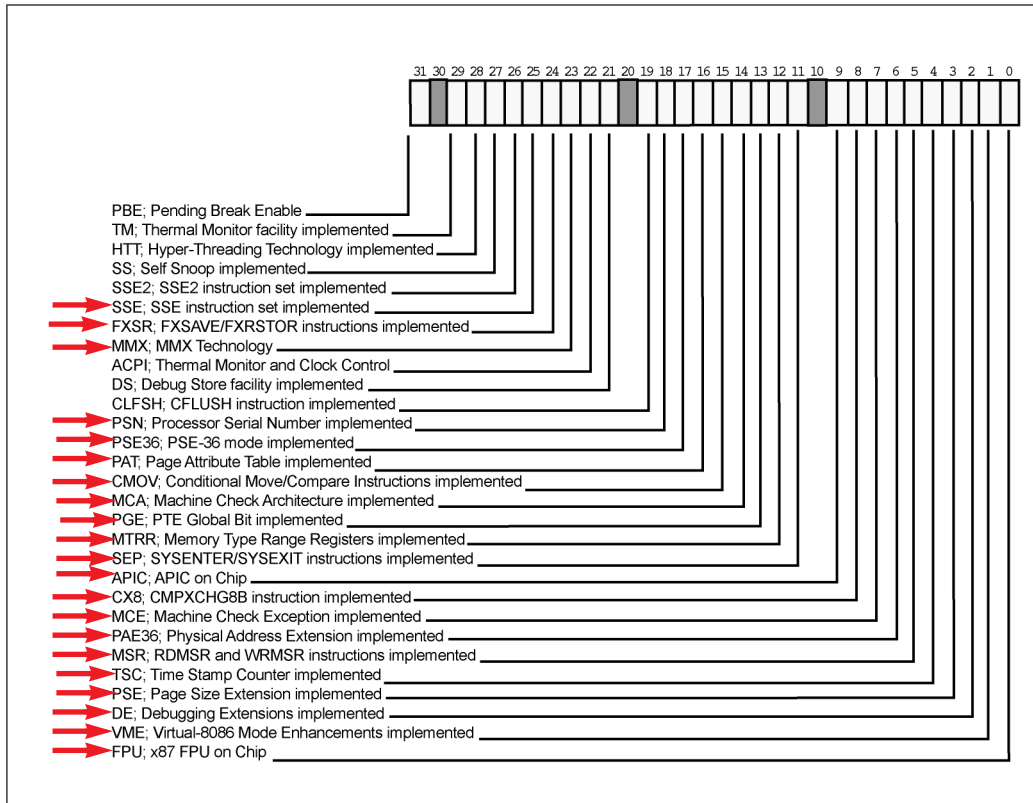
The Unabridged Pentium® 4

Table 3-6: Pentium® III Instruction Set Additions/Changes

Instruction	Description
CPUID	Enhanced to return the processor's serial number and Brand Index. See "CPU Identification" on page 1443 in the book for more information.
LDMXCSR	Used to read and write the MXCSR.
STMXCSR	
FXSAVE	Enhanced to save and restore the SSE registers as well as the FPU and MMX registers. In lieu of FXSAVE, the programmer could execute eight MOVDQ instructions to save the contents of XMM[7:0] to memory. See "FP/SSE Save/Restore Instruction Pair" on page 712 in the book for more information.
FXRSTOR	
MASKMOVQ	Store Selected Bytes of Qword to memory. See "The MASKMOVQ Instruction" on page 782 in the book for more information.
MOVNTQ	Qword Store Using Non-Temporal hint. See "The MOVNTQ Instruction" on page 781 in the book more information.
MOVNTPS	Store Packed SP FP Values Using Non-Temporal hint. See "The MOVNTPS Instruction" on page 780 in the book for more information.
PREFETCHH	Prefetch Data into Caches. See "Overlapping Data Prefetch with Program Execution" on page 773 in the book for more information.
SFENCE	Store Fence. See "The SFENCE Instruction" on page 784 in the book for more information.
SSE	SSE instruction set added. See "The Streaming SIMD Extensions (SSE)" on page 758 in the book for more information.

Chapter 3: Instruction Set Evolution

Figure 3-4: Pentium® III Capabilities



Pentium® 4 Additions to the Instruction Set

Table 3-7 on page 134 lists the new instructions that the Pentium® 4 added to the instruction set.

Figure 3-5 on page 135 illustrates the capabilities reported in the EDX register by a CPUID request type 1.

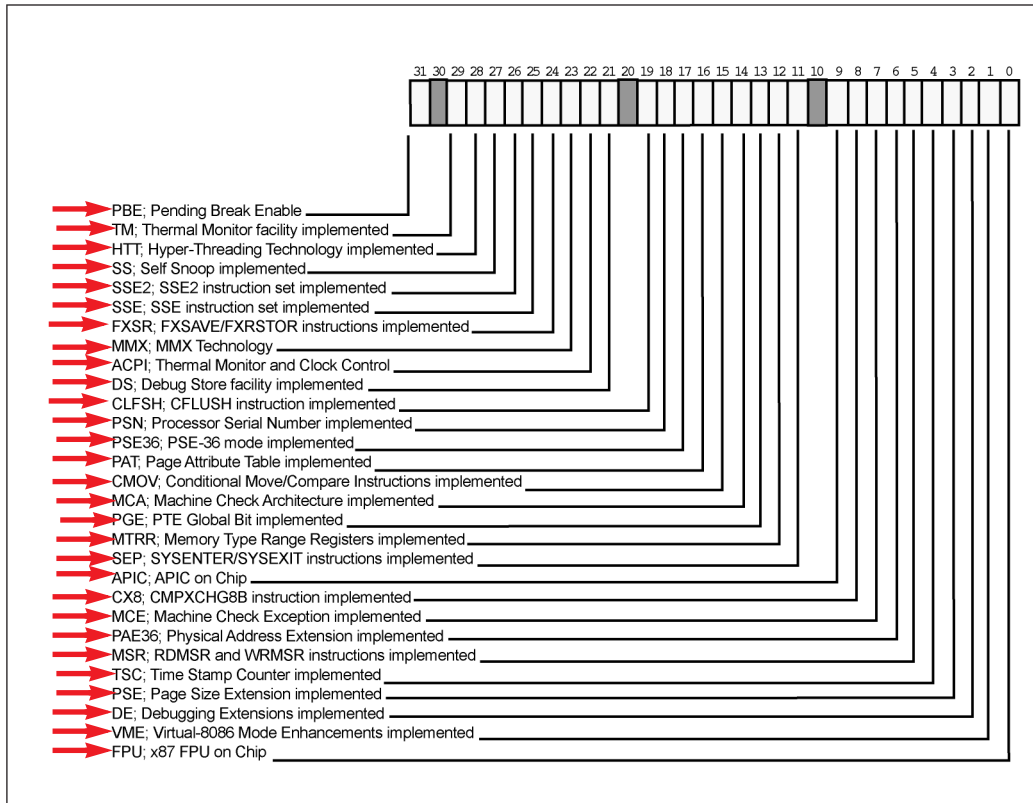
The Unabridged Pentium® 4

Table 3-7: Pentium® 4/Pentium® M Instruction Set Additions/Changes

Instruction	Description
CLFLUSH	Cache line flush. See “The Cache Line Flush Instruction” on page 1326 in the book for more information.
MFENCE	Memory fence. See “The Memory Fence Instruction” on page 1326 in the book for more information.
LFENCE	Load fence. See “The Load Fence Instruction” on page 1326 in the book for more information.
MOVNTDQ	Double qword store to memory using a non-temporal hint. See “The MOVNTDQ Instruction” on page 1327 in the book for more information.
MOVNTPD	Moves the two packed DP FP numbers from the source XMM register to memory using a non-temporal hint. See “The MOVNTPD Instruction” on page 1328 in the book for more information.
MOVNTI	Moves the source dword integer to memory using a non-temporal hint. See “The MOVNTI Instruction” on page 1329 in the book for more information.
MASK-MOVDQU	Stores selected bytes from the source XMM register into an 128-bit memory location. See “The MASKMOVDQU Instruction” on page 1330 in the book for more information.
PAUSE	See “The PAUSE Instruction” on page 1331 in the book for more information.
Branch hints	See “The Branch Hints” on page 1332 in the book for more information.
SSE2	SSE2 instruction set added. See “The SSE2 Instruction Set” on page 1332 in the book for more information.
SSE3	SSE3 instruction set added. See “The SSE3 Instruction Set” on page 1337 in the book for more information.

Chapter 3: Instruction Set Evolution

Figure 3-5: Pentium® 4/Pentium® M Capabilities



4

Exception Evolution

Introduction

The execution of instruction may result in an error condition. Upon detecting such an error condition, the processor generates a special type of interrupt referred to as an exception.

This chapter provides an overview of the evolution of the exception types generated by IA32 processors from the 386 through the Pentium® 4 processor.

Exception Evolution

Exceptions Defined For the 386

Table 4-1 on page 138 lists the exceptions defined for the 386.

Table 4-1: 386 Exceptions

Exception	Description
0	Divide-by-Zero Exception. A detailed description can be found in “Divide-by-Zero Exception (0)” on page 292 in the book.
1	Debug Exception. A detailed description can be found in “Debug Exception (1)” on page 293 in the book.
2	Non-Maskable hardware Interrupt (NMI). This is not an exception.
3	Caused by execution of the INT 3 (i.e., the Breakpoint) instruction. A detailed description can be found in “Breakpoint Exception (3)” on page 295 in the book.
4	Overflow Exception. Caused by EFlags[OF] detected set when the INTO instruction is executed. A detailed description can be found in “Overflow Exception (4)” on page 296 in the book.
5	Array Bounds Check Exception. Caused by the execution of the BOUND instruction. A detailed description can be found in “Array Bounds Check Exception (5)” on page 297 in the book.
6	Invalid OpCode Exception. Caused by the attempted execution of any invalid instruction. A detailed description can be found in “Invalid OpCode Exception (6)” on page 298 in the book.
7	Device Not Available (DNA) Exception. The Device-Not-Available exception is generated by any of three conditions: • The processor executed a FP instruction while CR0[EM] = 1. • The processor executed a WAIT or FWAIT instruction while CR0[MP] and CR0[TS] = 1, regardless of the state of CR0[EM]. • The processor executed a FPU instruction while CR0[TS] = 1 and CR0[EM] = 0. A detailed description can be found in “Device Not Available Exception (7)” on page 299 in the book.
8	Double Fault Exception. Indicates that the processor detected a second exception while calling an exception handler for a prior exception. Normally, when the processor detects another exception while trying to call an exception handler, the two exceptions can be handled serially. If, however, the processor cannot handle them serially, it signals the double-fault exception. A detailed description can be found in “Double Fault Exception (8)” on page 301 in the book.

Chapter 4: Exception Evolution

Table 4-1: 386 Exceptions (Continued)

Exception	Description
9	Coprocessor Segment Overrun. Indicates that a 386 system with a 387 coprocessor detected a page or segment violation while transferring the middle portion of a 387 coprocessor operand. A detailed description can be found in “Coprocessor Segment Overrun Exception (9)” on page 304 in the book.
10	Invalid TSS Exception. Occurs if, on the attempted execution of a JMP, CALL, IRET or INT instruction, the processor detects that the entry in the GDT selected by the instruction points to a TSS (Task State Segment) that contains invalid information. A detailed description can be found in “Invalid TSS Exception (10)” on page 305 in the book.
11	Segment Not Present Exception. When a new value is loaded into a segment register, the selected segment descriptor in the GDT or LDT has a zero in the Segment Present bit, indicating that the segment is not present in memory. A detailed description can be found in “Segment Not Present Exception (11)” on page 308 in the book.
12	Stack Exception. Caused by a stack overflow or underflow condition, or when a new value loaded into the SS register selects a segment that is not currently present in memory (i.e., the Segment Present bit in the segment descriptor is zero). A detailed description can be found in “Stack Exception (12)” on page 309 in the book.
13	General Protection (GP) Exception. The conditions that cause this exception to be generated comprise all of the protection violations that do not cause other exceptions to be generated (e.g., Invalid TSS, Segment-Not-Present, Stack Exception, or Page Fault Exceptions). A detailed description can be found in “General Protection (GP) Exception (13)” on page 311 in the book.

Table 4-1: 386 Exceptions (Continued)

Exception	Description
14	Page Fault Exception. Caused by any of the following conditions: • The P (present) bit in the Page Directory Entry (PDE) or Page Table Entry (PTE) selected by the 32-bit linear memory address = 0, indicating that either the selected Page Table or the target page is not currently present in physical memory. • The currently executing program does not have sufficient privilege to access the selected page. • The currently executing privilege level 3 program (i.e., an application program) attempted to write to a page marked as read-only by the selected PTE's attribute bits. • One or more reserved bits in the selected Page Directory Entry (PDE) = 1. A detailed description can be found in "Page Fault Exception (14)" on page 314 in the book.
15	Intel® Reserved.
16	FPU Exception. The x87 coprocessor detected an error while attempting execution of a FP instruction. A detailed description can be found in "FPU Exception (16)" on page 318 in the book.
17-31	Reserved.

486 Exception Additions/Changes

The 486 added or changed the exceptions listed in Table 4-2 on page 141.

Chapter 4: Exception Evolution

Table 4-2: New or Altered 486 Exceptions

Exception	Description
9	Was Coprocessor Segment Overrun. Changed to Reserved starting with the 486. Indicates that a 386 system with a 387 coprocessor detected a page or segment violation while transferring the middle portion of a math coprocessor operand. Starting with the 486, IA32 processors do not generate this exception. Rather, the same condition is reported via the General Protection Exception (Exception 13). A detailed description can be found in “Coprocessor Segment Overrun Exception (9)” on page 304 in the book.
17	Alignment Check Exception. Was Reserved prior to the 486. Indicates that the processor detected an access attempt to a misaligned memory operand while alignment checking was enabled. Alignment checks are only carried out in data and stack segments, never in code or system segments. A detailed description can be found in “Alignment Check Exception (17)” on page 321 in the book.

Pentium® Exception Additions/Changes

The Pentium® added the exceptions listed in Table 4-3 on page 141.

Table 4-3: New or Altered Pentium® Exceptions

Exception	Description
13	A GP exception occurs when an attempt is made to write a one into a reserved bit position of a special register (i.e, a Control register). A detailed description can be found in “General Protection (GP) Exception (13)” on page 311 in the book.
14	A Page Fault exception occurs when a one is detected in any of the reserved bit positions of a Page Table Entry (PTE) or a Page Directory Entry (PDE) during linear-to-physical address translation. A detailed description can be found in “Page Fault Exception (14)” on page 314 in the book.

The Unabridged Pentium® 4

Table 4-3: New or Altered Pentium® Exceptions (Continued)

Exception	Description
18	Machine Check Exception. Used for reporting parity errors and other hardware errors. CR4[MCE] is the Machine Check Enable bit. When MCE = 0, the processor inhibits generation of the Machine Check Exception. If the Machine Check interrupt entry in the IDT (Interrupt Descriptor Table) is being used for another purpose and CR4[MCE] = 1, the interrupt routine executed must examine the state of the CHK bit in P5_MC_TYPE MSR to determine the cause of the interrupt (this only applies to the Pentium). A detailed description can be found in “Machine-Check Exception (18)” on page 324.

Pentium® Pro and Pentium® II Exception Additions/Changes

There were no new exceptions, nor were any pre-existent exceptions altered.

Pentium® III

The Pentium® III added the exception listed in Table 4-4 on page 142.

Table 4-4: New Pentium® III Exceptions

Exception	Description
19	SIMD FP Exception was added. Indicates that the processor has detected an error while attempting execution of an SSE or SSE2 SIMD FP instruction. A detailed description can be found in “SIMD Floating-Point Exception (19)” on page 326 in the book.

Pentium® 4/Pentium® M Exception Additions/Changes

There were no new exceptions, nor were any pre-existent exceptions altered.

5

P6 Road Map

Introduction

This chapter provides an introduction to the P6 processor family.

The P6 Processor Family

All Intel® IA32 processors in the Pentium® Pro, Pentium® II and Pentium® III product lines (including Celerons and Xeons) are referred to as P6 family processors because they were all based on variants of the P6 processor core. The three core variants were code named (in chronological order) as follows:

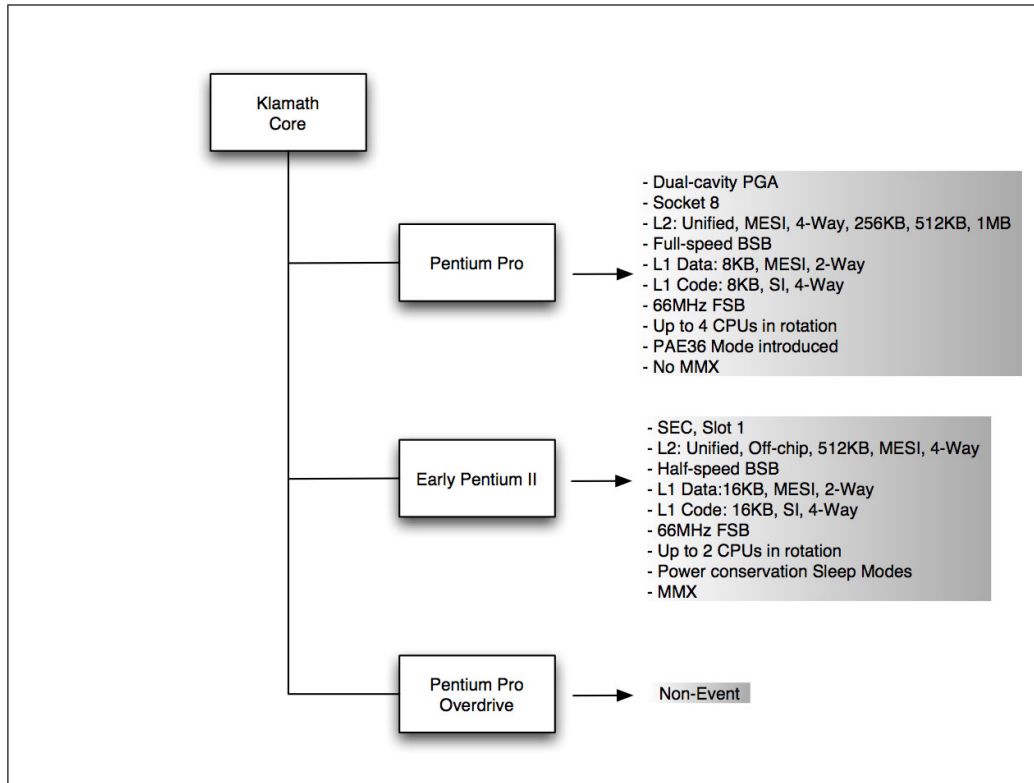
- The Klamath core.
- The Deschutes core.
- The Katmai core.

The next three sections provide an overview of these three cores and the products that were based on each of them.

The Klamath Core

Figure 5-1 on page 144 illustrates (in chronological order) the Intel® products that were based on the Klamath processor core.

Figure 5-1: P6 Klamath Core Roadmap



Pentium® Pro Processor

The basic characteristics of the Pentium® Pro processor are described in the following subsections.

Dual-Cavity PGA Package

It was produced using a unique dual-cavity PGA package (see Figure 5-2 on page 151) with the processor core (including the L1 Code and Data caches) on one die and the L2 Cache on the other die. Due to the proximity of the two dies, the BSB connecting the two dies could be operated at the full processor core clock speed (which ultimately reached a top speed of 200MHz).

The PGA package plugged into Socket 8 on the system board.

Dedicated L2 Cache

The L2 Cache has the following basic characteristics:

- It is a unified cache that keeps copies of both code lines and data lines to service requests issued by the two L1 caches.
- MESI. The state of a line in the L2 Cache could be:
 - Invalid (I): The L2 Cache does not have a copy of the line. Other processors may or may not have a copy of the line.
 - Exclusive (E): The L2 Cache has an exclusive copy of the line (i.e., no other processor has a copy of the line) and the copy is the same as the line in system memory.
 - Shared (S): The L2 Cache has a copy of the line and at least one other processor also has a cached copy of the line. The cached copy of the line is still the same as the line in system memory.
 - Modified (M): The L2 Cache has a copy of the line that has received one or more updates (i.e., memory writes) by the program executing on the processor core since the line was read from memory. No other processor has a copy of the line. The copy of the line in system memory is stale (i.e., it has not received the updates).
- 4-Way Set-Associative. The L2 Cache implements four banks (aka Ways) of high-speed static RAM that are used to cache copies of lines read from system memory. The cache also implements four directories, one associated with each of the data storage banks. Each directory entry keeps track of the page in memory that the line was read from as well as the current state of the line (i.e., M, E, S or I).
- The overall size of the four data storage banks was available in the following sizes:
 - 256KB organized as four banks of 64KB each, each bank divided into 2048 lines of 32 bytes each.
 - 512KB organized as four banks of 128KB each, each bank divided into 4096 lines of 32 bytes each.
 - 1MB organized as four banks of 256KB each, each bank divided into 8192 lines of 32 bytes each.

BSB Speed and Width

The BSB connecting the core to the L2 Cache ran at the full processor core speed. A 64-bit data bus connected the L2 Cache to the L1 Code and Data Caches.

L1 Data Cache

The L1 Data Cache has the following basic characteristics:

- It only caches data lines obtained from system memory.
- It services memory data reads (i.e., loads) and writes (i.e., stores) executed by the processor core.
- 2-Way Set-Associative. The L2 Cache implements two banks (aka Ways) of high-speed static RAM that are used to cache copies of data lines read from system memory. The cache also implements two directories, one associated with each of the data storage banks. Each directory entry keeps track of the page in memory that the line was read from as well as the current state of the line (i.e., M, E, S or I).
- MESI. The state of a line in the L1 Data Cache could be M, E, S or I.
- The L1 Data Cache size is 8KB, 2-way set-associative organized as two banks of 4KB each, each bank divided into 128 lines of 32 bytes each.

L1 Code Cache

The L1 Code Cache has the following basic characteristics:

- It only caches code lines obtained from system memory.
- It services memory instruction reads executed by the processor core's instruction prefetcher.
- 4-Way Set-Associative. The L2 Cache implements four banks (aka Ways) of high-speed static RAM that are used to cache copies of code lines read from system memory. The cache also implements four directories, one associated with each of the code storage banks. Each directory entry keeps track of the page in memory that the line was read from as well as the current state of the line (i.e., S or I).
- SI. The L1 Code Cache implements a subset of the MESI protocol. The state of a line in the L1 Code Cache could be S or I. The Code cache is only read from (by the instruction prefetcher) and is never written to.
- The L1 Code Cache size is 8KB, 4-way set-associative organized as four banks of 2KB each, each bank divided into 64 lines of 32 bytes each.

FSB Speed

The FSB operational speed is 66MHz. All FSB transactions are synchronized to the 66MHz BCLK (Bus Clock) signal that is routed to all devices that reside on the FSB.

FSB Protocol

The FSB protocol implemented on the IA32 processors prior to the Pentium® Pro was very simple and designed for a system with only one processor residing on the FSB. The protocol was not well-suited to a system design wherein more than one processor resided on the FSB.

The FSB protocol utilized on the P6 family processors was specifically designed to efficiently handle a number of processors as well as additional devices residing on the same FSB.

FSB Arbitration Scheme

The FSB arbitration scheme supports up to four CPUs using a strict rotational algorithm. Processors are referred to as symmetric bus agents because they use a symmetric, rotational bus arbitration scheme. In addition, devices other than processors may need to initiate transactions on the FSB. These devices are referred to as Priority Agents. When such a device requires the use of the FSB to perform a transaction, it asserts a signal to the community of processors requesting that it be permitted to break into the rotation in order to initiate its transaction.

Address Bus Width

The address bus width is 36 bits, permitting up to 64GB of memory to be addressed.

PAE-36 Mode

This was the first IA32 processor to implement PAE-36 Mode (PAE = Physical Address Extension).

- If the OS implements 3-level rather than 2-level Paging address translation, the OS sets CR4[PAE] = 1. This permits the Paging mechanism to translate 32-bit linear memory addresses into 36-bit physical memory addresses (and therefore the processor can address the full 64GB memory address range).
- If CR4[PAE] = 0, the OS implements the 2-level, 386-compatible Page address translation mechanism and can only translate 32-bit linear memory addresses into 32-bit physical memory addresses. In other words, the processor could not address memory above the 4GB address boundary.

No MMX

The Pentium® Pro processor was intended for the server market. Since servers traditionally did not have a need for high-end graphics and multimedia, the processor did not implement the MMX instruction set. It should be noted that with the exception of the Pentium® Pro, all IA32 processors starting with the P55C Pentium® include MMX capability.

Power Conservation States

The Pentium® Pro supported the following power conservation modes:

- Normal state. The processor is running at the full processor clock speed and no power conservation is in effect.
- AutoHalt Power-Down state.
- Stop Grant state.
- Halt/Grant Snoop state.

All four of these states are implemented in all subsequent IA32 processors. Each of the three power conservation states are described in the subsections that follow.

AutoHalt Power Down State. When a HLT instruction is executed, the processor generates a Special transaction on its FSB to broadcast a Halt message to the system (see “The Special Transaction” on page 1306 in the book). It then leaves the Normal state and enters the AutoHalt Power Down state. This state has the following characteristics:

- The processor powers down all logic not necessary for the recognition of interrupts and the snooping of memory accesses generated by other entities on the FSB.
- The BCLK signal on the FSB continues to run.
- The processor snoops any memory accesses initiated by other FSB entities and then returns to the AutoHalt Power Down state. To do this, the processor transitions from the AutoHalt Power Down state to the Halt/Grant Snoop state (see “Halt/Grant Snoop State” on page 150). After the snoop is complete, the processor returns to the AutoHalt Power Down state.
- Upon the occurrence of an interrupt event (RESET#, SMI#, BINIT#, INIT#, or LINT[1:0]—NMI or INTR), the processor exits the AutoHalt Power Down state and returns to the Normal state to service the interrupt.
- Upon return from the SMI interrupt handler, the processor either enters the Normal state (if the instruction returned to is an instruction other

than a HLT) or to the AutoHalt Power Down state (if the instruction returned to is a HLT instruction).

- If the processor's FLUSH# is asserted while in the AutoHalt Power Down state, the flush is serviced (i.e., all modified lines are written back to system memory and the caches are invalidated). Upon completion of the writeback operation, the processor re-enters the AutoHalt Power Down state.
- The system board logic may assert the STPCLK# signal to the processor while the processor is in the AutoHalt Power Down state. This causes the processor to leave the AutoHalt Power Down state and enter the Stop Grant state (see the next subsection). The processor remains in the Stop Grant state until STPCLK# is removed and then re-enters the AutoHalt Power Down state.

Stop Grant State. Entered from the following states:

- Entered from the Normal state when the chipset asserts STPCLK# to the processor.
- Entered from the AutoHalt Power Down state when the chipset asserts STPCLK# to the processor while the processor is in the AutoHalt Power Down state.

The Stop Grant state has the following characteristics:

- Upon entering the Stop Grant state from the Normal state, the processor generates a Special transaction to broadcast a Stop Grant Acknowledge message on the FSB (see "The Special Transaction" on page 1306 in the book).
- The processor powers down all logic unnecessary for the recognition of interrupts and the snooping of memory accesses generated by other FSB agents.
- To ensure minimum power consumption, the system board logic ensures that the FSB GTL+ signals (see "Everything's Relative" on page 1127 in the book) are not driven low during the Stop Grant period. In addition, all other processor input signals on the FSB should be deasserted.
- The BCLK signal on the FSB continues to run.
- The processor snoops any memory accesses initiated by other FSB entities and then returns to the AutoHalt Power Down state. To do this, the processor transitions from the AutoHalt Power Down state to the Halt/Grant Snoop state (see "Halt/Grant Snoop State" on page 150). After the snoop is complete, the processor returns to the AutoHalt Power Down state.

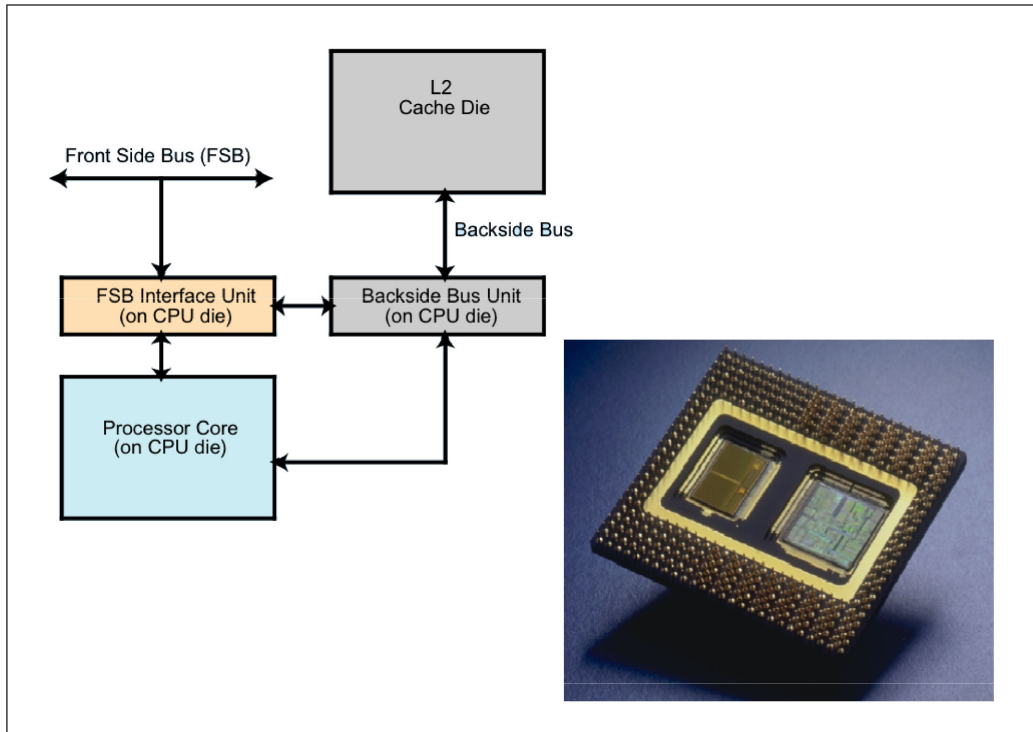
- Upon the occurrence of some interrupt events (SMI#, INIT#, or LINT[1:0]—NMI or INTR), the processor latches the event. It is not serviced until the processor returns to the Normal state (when STPCLK# is deasserted), however. Note that the processor can only latch one of each of these events while in the Stop Grant state.
- If the processor's FLUSH# input is asserted while the processor is in the Stop Grant state, the flush is serviced (i.e., all modified lines are written back to system memory and the caches are invalidated). Upon completion of the writeback operation, the processor re-enters the Stop Grant state.
- The assertion of its RESET# input causes the processor to return to the initialized state.
- A transition to the Sleep state occurs if the processor's SLP# input is asserted while the processor is in the Stop Grant state.

Halt/Grant Snoop State. The processor snoops all memory transactions generated on the FSB by other FSB agents during both the AutoHalt Power Down and the Stop Grant states. When the Snoop Phase of a memory transaction is detected, the processor temporarily transitions to the Halt/Grant Snoop state, snoops the transaction in its caches, and then returns to the state that the Halt/Grant Snoop state was entered from.

Dual-Independent Bus Architecture (DIBA)

The Pentium® Pro processor and all subsequent IA32 processors have a dedicated backside bus used to transfer data between the processor core and the L2 Cache and an external, or front side, bus used to communicate with other system devices. This architecture is sometimes referred to as DIBA.

Figure 5-2: Pentium® Pro Dual-Cavity PGA Package



Early Pentium® II Processor

The basic characteristics of the early Pentium® II processor are described in the following subsections.

Processor Cartridge and Slot

At the time that the Pentium® Pro and Pentium® II processors were developed, the die technologies available did not permit the L2 Cache to be incorporated on the processor die. The Pentium® Pro design solved this problem by using the dual-cavity design, but this technique was expensive to implement. The designers of the Pentium® II used another approach. The processor core was incorporated on a circuit card along with the L2 Cache. This card was then plugged into an in-line connector on the system board referred to as the Slot 1 connector.

Figure 5-3 on page 157 illustrates one side of the card showing the processor core and two of the four banks of the L2 Cache. Figure 5-4 on page 157 shows

the other side of the card containing the other two L2 Cache banks, as well as the L2 Tag SRAM (i.e., the 4 directories associated with the four L2 Cache banks). Figure 5-5 on page 158 shows the card enclosed in its cover. The entire assembly was referred to as an SEC (Single-Edge Contact) cartridge.

Figure 5-6 on page 158 illustrates a simplified Pentium® II block diagram consisting of the following elements:

- The processor core. This includes the L1 Data and L1 Code Caches, as well as the Local APIC module.
- The 3-wire APIC bus connects the processor's Local APIC through the edge connector to the APIC bus on the system board.
- The Bus Interface Unit connects the processor core to the FSB on the system board through the edge connector.
- The on-card BSB connects the processor core to the L2 Cache.

The L2 Cache

The L2 Cache has the following basic characteristics:

- It is a unified cache that keeps copies of both code lines and data lines to service requests issued by the two L1 caches.
- MESI. The state of a line in the L2 Cache could be:
 - Invalid (I): The L2 Cache does not have a copy of the line. Other processors may or may not have a copy of the line.
 - Exclusive (E): The L2 Cache has an exclusive copy of the line (i.e., no other processor has a copy of the line) and the copy is the same as the line in system memory.
 - Shared (S): The L2 Cache has a copy of the line and at least one other processor also has a cached copy of the line. The cached copy of the line is still the same as the line in system memory.
 - Modified (M): The L2 Cache has a copy of the line that has received one or more updates (i.e., memory writes) by the program executing on the processor core since the line was read from memory. No other processor has a copy of the line. The copy of the line in system memory is stale (i.e., it has not received the updates).
- 4-Way set-associative. The L2 Cache implements four banks (aka Ways) of high-speed static RAM that are used to cache copies of lines read from system memory. The cache also implements four directories, one associated with each of the data storage banks. Each directory entry keeps track of the page in memory that the line was read from as well as the current state of the line (i.e., M, E, S or I).
- The overall size of the four data storage banks was 512KB organized as four banks of 128KB each, each bank divided into 4096 lines of 32 bytes each.

The BSB Speed and Width

The BSB connecting the core to the L2 Cache ran at one half the processor core speed. The longer trace runs interconnecting the components on the card necessitated running the BSB at a slower speed.

A 64-bit data bus connected the L2 Cache to the L1 Code and Data Caches.

The L1 Data Cache

The L1 Data Cache differs from that implemented on the Pentium® Pro in only one respect: the cache size was doubled from 8KB to 16KB to compensate for the slower BSB speed. The 2-way set-associative L1 Data Cache is organized as two banks of 8KB each, each bank divided into 256 lines of 32 bytes each. In all other respects, the L1 Data Cache is identical to that implemented in the Pentium® Pro (see “L1 Data Cache” on page 146).

The L1 Code Cache

The L1 Code Cache differs from that implemented on the Pentium® Pro in only one respect: the cache size was doubled from 8KB to 16KB to compensate for the slower BSB speed. The 4-way set-associative L1 Code Cache is organized as four banks of 4KB each, each bank divided into 128 lines of 32 bytes each. In all other respects, the L1 Code Cache is identical to that implemented in the Pentium® Pro (see “L1 Code Cache” on page 146).

The FSB Speed

The FSB operational speed is 66MHz. All FSB transactions are synchronized to the 66MHz BCLK (Bus Clock) signal that is routed to all devices that reside on the FSB.

The FSB Arbitration Scheme

The FSB arbitration scheme is identical to that of the Pentium® Pro with one exception: a maximum of two processors is supported (rather than four for the Pentium® Pro). See “FSB Arbitration Scheme” on page 147 for more information.

Address Bus Width

The address bus width is 36 bits, permitting up to 64GB of memory to be addressed.

Power Conservation States

The Pentium® II supported the following power conservation modes (see “Power Conservation States” on page 148 for a description of the first four modes):

- Normal state.
- AutoHalt Power-Down state.
- Stop Grant state.
- Halt/Grant Snoop state.
- Sleep state.
- Deep Sleep state.

The Sleep and Deep Sleep power conservation states were added in the Pentium® II, are present in all subsequent IA32 processors, and are described in the subsections that follow.

Sleep State. The processor enters the Sleep state only from the Stop Grant state. This occurs when the processor’s SLP# (Sleep) input is asserted while the processor is in the Stop Grant state. Upon entering the Sleep state, the processor powers off much of its logic, including that necessary for:

- snooping external memory accesses and
- latching interrupt events.

The Sleep state has the following characteristics:

- The processor maintains the contents of its registers and caches.
- The processor’s PLL (Phase-Locked Loop) continues to run, generating the processor internal clock.
- All internal clocks are stopped. This stops CMOS input receivers from switching, thereby diminishing power consumption.
- The SLP# pin is not recognized during the Normal or AutoHalt states.
- Snoop events (i.e., memory transactions initiated by other FSB agents) that occur on the FSB while the processor is in the Sleep state or while the processor is transitioning in or out of the Sleep state result in unpredictable behavior.
- No transitions or assertions of signals (with the exception of SLP# or RESET#) are allowed on the FSB while the processor is in the Sleep state. Any transitions on any input signal before the processor has returned to the Stop Grant state result in unpredictable behavior.

- If RESET# is asserted while the processor is in the Sleep state, the processor is initialized and immediately transitions to the Normal state (without passing through the Stop Grant state). The SLP# and STP-CLK# processor inputs must be deasserted immediately after its RESET# input is asserted.
- The SLP# pin has a minimum assertion period of one BCLK (Bus Clock).

Deep Sleep State. This is the lowest power conservation state that the processor can be in and still maintain the contents of its registers and caches. The Deep Sleep state has the following characteristics:

- It can only be entered after the processor has entered the Sleep state.
- The processor enters the Deep Sleep state immediately when the system board logic turns off the BCLK input to the processor. Intel® recommends that the BCLK be stopped in the low state, lowering current consumption to leakage levels.
- To exit the Deep Sleep state and return to the Sleep state, the BCLK must be restarted by the system board logic and must continue to run for a period of 1ms.
- Snoop events (i.e., memory accesses initiated by other FSB agents) that occur on the FSB while the processor is in the Deep Sleep state result in unpredictable behavior.
- No transitions or assertions of signals are allowed on the FSB while the processor is in the Deep Sleep state. Any transitions on any input signal before the processor has returned to the Stop Grant state result in unpredictable behavior.

MMX

The Pentium® Pro processor did not implement the MMX execution units or the MMX register set and therefore did not support the MMX instruction set. The Pentium® II processor, on the other hand, implements MMX. A description of MMX can be found in “MMX Capability” on page 519 in the book.

With the exception of the MMX Multiplier execution unit which was pipelined and had a latency of three processor clock cycles, the other MMX execution units had a latency of one processor clock cycle. The Pentium® II's context switching times between FP and MMX modes was improved over that demonstrated by the Pentium® P55C MMX processor.

Enhanced Legacy Code Execution Speed

The Pentium® Pro processor design was optimized for 32-bit code execution. One of the characteristics of 32-bit code is that the data segment registers are almost never loaded with new values. 16-bit code, however, writes new values into the data segment registers (DS, ES, FS, and GS) quite frequently.

Unlike the processor's GPRs, the Pentium® Pro processor's data segment registers were not aliased. A write to a data segment register immediately changed the value in the targeted data segment register. Instructions that resided downstream from the instruction that changes the data segment register might perform accesses (loads or stores) within the affected data segment. If the processor were permitted to execute instructions beyond the one that changed the data segment register before that instruction had completed execution, they would be using the old, stale contents of the segment register and would address the wrong location.

For this reason, the Pentium® Pro processor did not execute any instructions beyond the segment register load instruction until the load had completed execution. The processor's performance degraded because it was restrained from out-of-order execution. *Since 16-bit code changed the contents of the data segment registers a lot, the processor suffered poor performance when executing 16-bit code.* Since Windows 95 implemented quite a bit of 16-bit code, the Pentium® Pro's performance suffered when executing 16-bit code.

Intel® fixed this problem in the Pentium® II processor by aliasing (or shadowing) the data segment registers (as was already done for the GPRs). This was accomplished in the RAT stage of the instruction pipeline. Intel® estimated that the Pentium® II processor's shadowing of the segment registers could increase performance when executing 16-bit code by as much as 8 to 10%.

Figure 5-3: SEC Package (1-of-2)

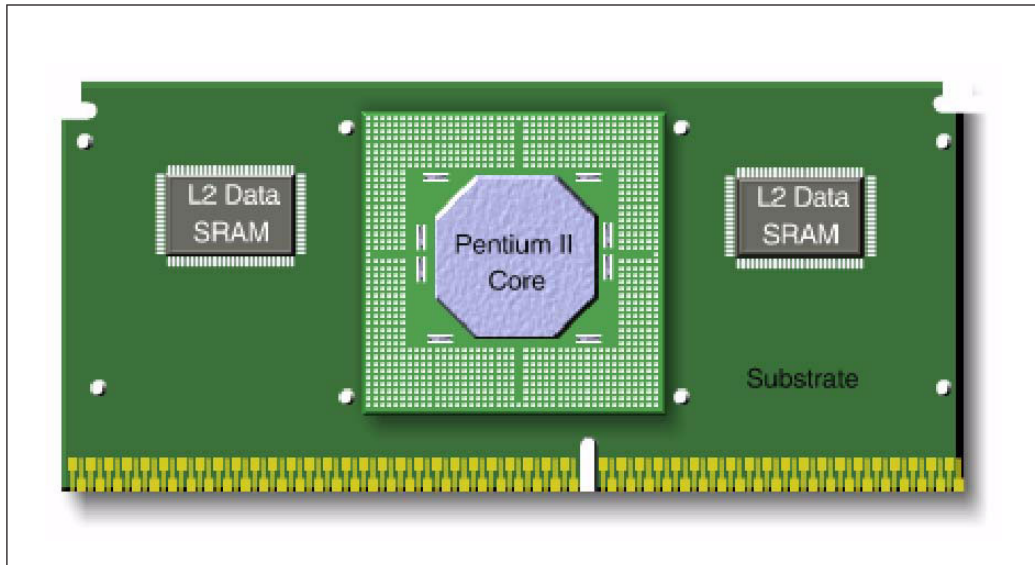
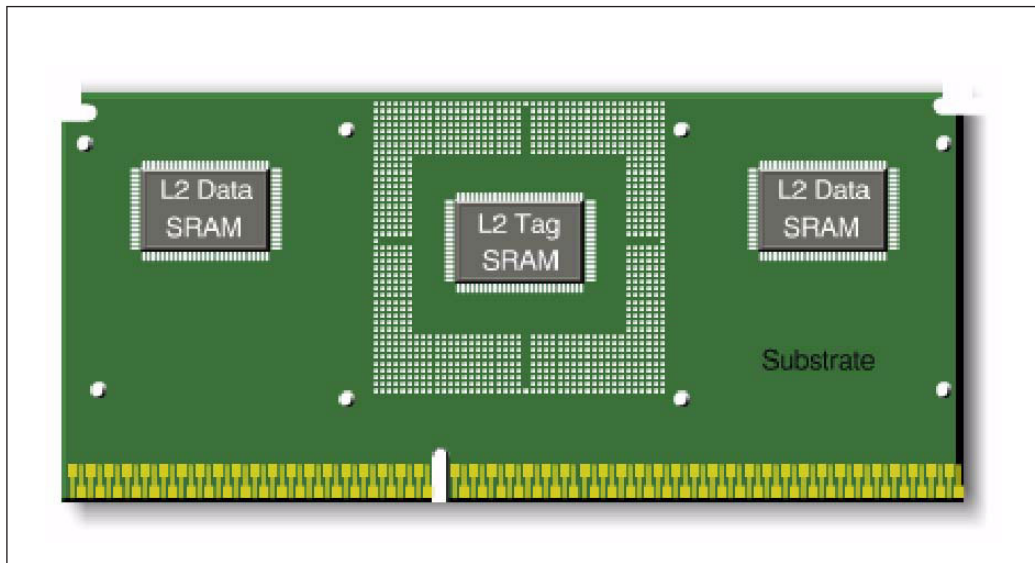


Figure 5-4: SEC Package (2-of-2)

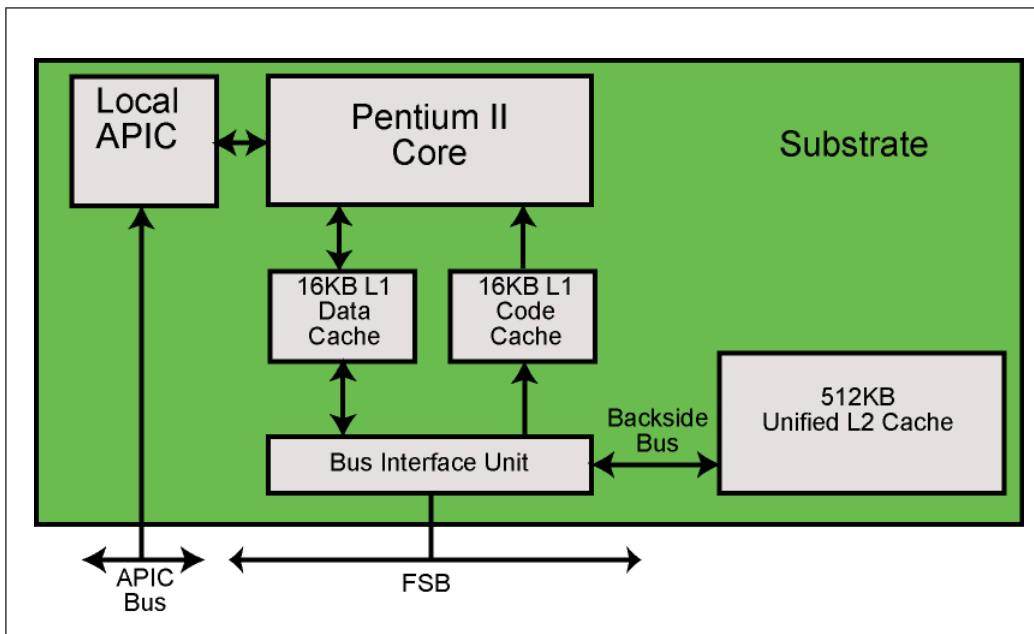


The Unabridged Pentium® 4

Figure 5-5: SEC Cartridge Cover



Figure 5-6: Pentium® II Block Diagram (Simplified)



Pentium® II Overdrive Processor

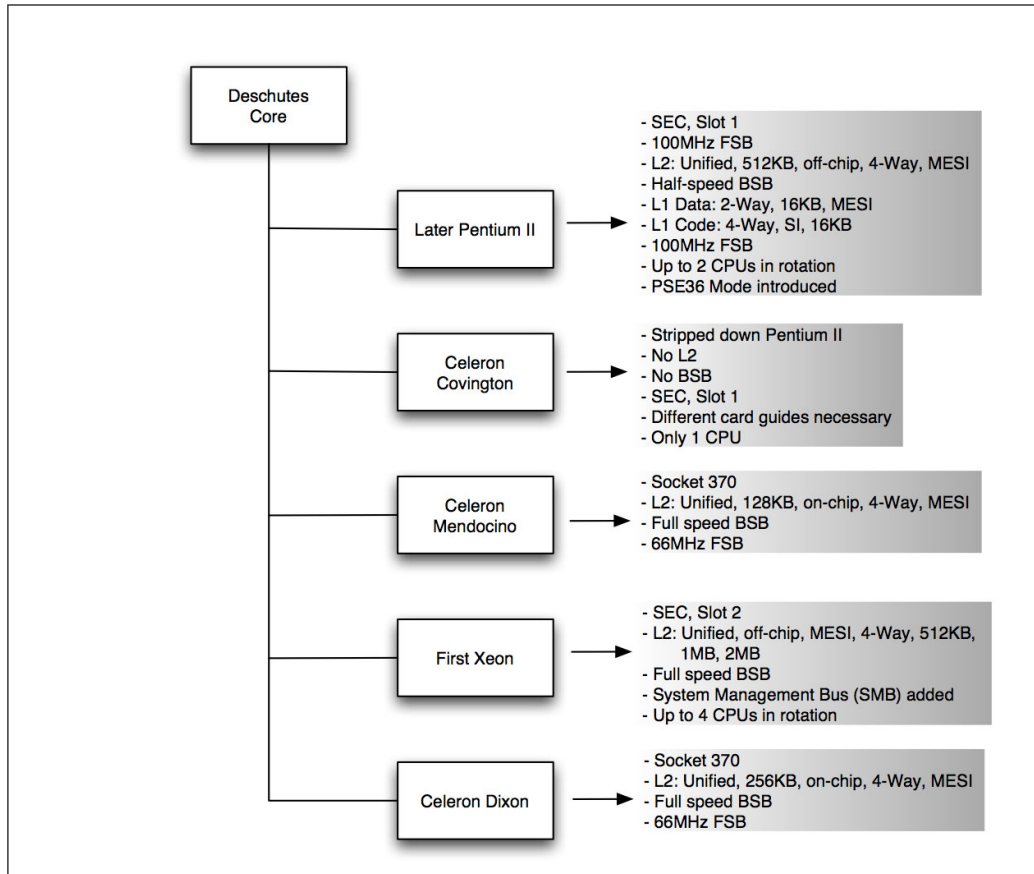
Basically, the Pentium® II Overdrive processor is a pin-compatible Pentium® II replacement for the Pentium® Pro processor. As stated earlier, the Pentium® Pro processor did not implement MMX. In addition, the Pentium® Pro was not very good at executing legacy 16-bit code (see “Enhanced Legacy Code Execution Speed” on page 156). The Pentium® II Overdrive provided the following enhancements:

- The Pentium® II OverDrive processor could upgrade 150- and 180MHz Pentium® Pro processor-based systems to 300MHz, and 166- and 200MHz processor-based systems to 333MHz.
- Added MMX support.
- Doubled the size of the L1 caches from 8KB to 16KB.
- Improved execution speed when executing legacy 16-bit code.

The Deschutes Core

Figure 5-7 on page 160 illustrates (in chronological order) the Intel® products that were based on the Deschutes processor core.

Figure 5-7: P6 Deschutes Core Roadmap



Later Pentium® II Processor

The later model Pentium® II processor had the following basic characteristics that differentiated it from the earlier Pentium® II processor:

- The FSB speed was increased from 66MHz to 100MHz.
- PSE-36 Mode was introduced. This is an alternative to PAE-36 Mode. A detailed explanation can be found in “PSE-36 Mode” on page 731 in the book.

Celeron Covington Processor

At a certain point in time, Intel® found that it was losing market share to AMD in the market for small footprint systems and laptops. At the time, the Pentium® II processor was the only solution Intel® had to offer. It was bulky (due to the cartridge format) and expensive.

As a short term solution, Intel® produced a stripped down version of the Pentium® II processor by removing the L2 Cache. It was the first Intel® processor branded with the Celeron name and was code named “Covington.” All processors that are members of the Celeron brand only support one processor on the FSB.

Unfortunately, its performance was not very good due to the missing L2 Cache.

Celeron Mendocino Processor

The first viable member of the Celeron brand was code named “Mendocino.” It had the following characteristics that differentiated it from a Pentium® II processor:

- It was the first Intel® processor with the L2 Cache integrated onto the processor die. The L2 Cache had the following characteristics:
 - Unified code/data cache.
 - 128KB.
 - 4-Way set-associative.
 - MESI.
- The BSB ran at the full processor core speed. This yielded an interesting side effect: when executing some applications (where the code and/or data being operated upon fit nicely in the L2 Cache) this Celeron could out perform the higher-cost Pentium® II processor.
- In order to differentiate it from the higher-cost Pentium® II, the FSB ran at 66MHz (rather than 100MHz for the Pentium® II). Even though the BSB speed permitted the L2 Cache to be accessed quickly, the slower FSB speed was an artificial way of throttling the Celeron’s performance.
- With the advent of this processor, the processor size (which now included the L2 Cache) had been reduced to the point where Intel® could revert back to socketed processors rather than using the cartridge format. This processor was installed in Socket 370.

Pentium® II Xeon Processor

The Pentium® II Xeon processor was the first Xeon and had the following basic characteristics:

- Although it also used the cartridge format, the Xeon cartridge was larger than the Pentium® II cartridge.
- The cartridge installed into the Slot 2 connector on the system board. Slot 2 differed from Slot 1 in that it included a larger number of power and ground pins (due to the larger power requirements necessitated by the larger L2 Cache).
- The BSB ran at the full processor core speed. This is also true of all subsequent Xeon processors.
- The L2 Cache was not on the processor die, but rather was mounted on the cartridge substrate (as it was on the Pentium® II processor).
- The L2 Cache was available in sizes of 512KB, 1MB and 2MB and was a unified, 4-way set-associative MESI cache.
- This was the first Intel® processor to implement the System Management Bus (SM Bus). All Xeon processors implement the SMBus.
- The processor included the following internal entities that can only be accessed over the SMBus:
 - An internal thermal diode.
 - A 128-byte Processor Information ROM (PIROM) containing information about the processor.
 - A 128-byte scratch EEPROM that can be used by the system vendor to store information.
- The FSB arbitration scheme on most Xeon processors supports up to four processors on the FSB. Some later model Xeon processors with the “DP” (dual processor) designation at the end of the name only support up to two processors.

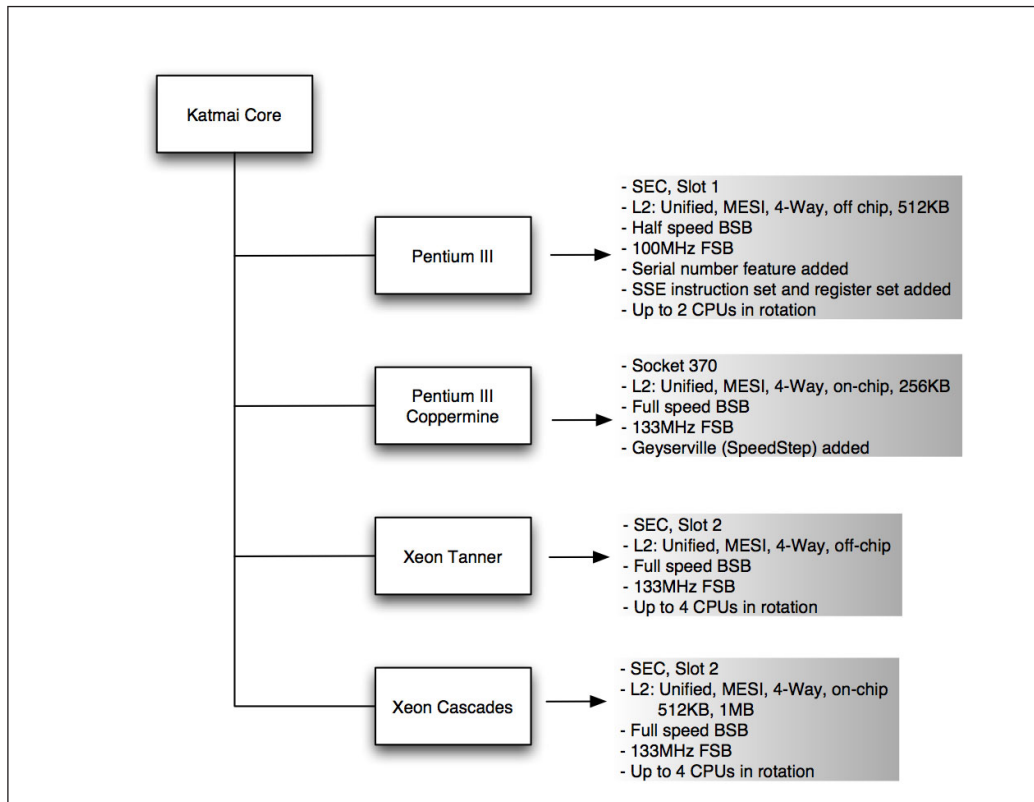
Celeron Dixon Processor

The Celeron Mendocino was the first Intel® processor with the L2 Cache integrated onto the processor die. While its L2 Cache size was 128KB, the Celeron Dixon processor integrated a 256KB L2 Cache onto the processor die. Like the L2 Caches on previous P6 family processors, it was a unified, 4-Way set-associative, MESI cache. The BSB ran at the full processor core clock speed.

The Katmai Core

Figure 5-8 on page 163 illustrates (in chronological order) the Intel® products that were based on the Katmai processor core. Basically, all versions of the Pentium® III processor were based on the Katmai core.

Figure 5-8: P6 Katmai Core Roadmap



Early Pentium® III Processor

The early model Pentium® III processor had the following basic characteristics:

- It was implemented using the SEC cartridge format and installed into the Slot 1 connector on the system board.
- The L2 Cache was not integrated onto the processor die, but rather was implemented as discrete SRAM chips on the cartridge substrate. The L2 Cache was a unified, MESI, 4-Way set-associative, 512KB cache.
- It had a half-speed BSB.
- The FSB ran at 100MHz.
- It implemented the ill-fated Serial Number feature. Due to the great outcry from privacy advocates, this feature was subsequently disabled and later removed.
- The SSE instruction set and register set was added. Note that all processors built on the Katmai core include SSE.
- The FSB arbitration scheme supported one or two processors in a rotational bus arbitration scheme.

Pentium® III Coppermine Processor

The later model Pentium® III processor (code name “Coppermine”) had the following basic characteristics:

- The 256KB L2 Cache was integrated onto the processor die. The L2 Cache was a unified, 4-way set-associative cache.
- The processor plugged into Socket 370.
- It implemented a full-speed BSB.
- The speed of the FSB was bumped up to 133MHz.
- The mobile version of “Coppermine” was the first processor to implement SpeedStep Technology (code named “Geyserville”). SpeedStep’s original implementation was fairly primitive. The processor dynamically switched clock frequency and voltage, depending on whether the system was running on batteries or AC power. When running in Battery Optimized Mode, the processors run at a lower frequency and lower voltage (1.35 volts), lowering CPU power consumption as well as performance. When the system was plugged into AC power, the processor automatically switched to Maximum Performance Mode, increasing the processor’s internal clock frequency. The mode switch took approximately 1/2000th of a second.

Xeon Tanner Processor

The Pentium® III Xeon Tanner processor had the following basic characteristics:

- Using the Xeon cartridge format, it installed into the Slot 2 connector on the system board.
- The L2 Cache was not integrated onto the processor die, but rather was implemented as discrete SRAM chips on the cartridge substrate. The L2 Cache was a unified, MESI, 4-Way set-associative cache.
- It had a full-speed BSB.
- It had a 133MHz FSB.
- The FSB arbitration scheme supported up to four processors in a rotational bus arbitration scheme.
- The processor included the following internal entities that can only be accessed over the SM Bus:
 - An internal thermal diode.
 - A 128-byte Processor Information ROM (PIROM) containing information regarding the processor.
 - A 128-byte scratch EEPROM that can be used by the system vendor to store information.

Xeon Cascades Processor

The Pentium® III Xeon Cascades processor differed from the Xeon Tanner in that it was the first Xeon processor with the L2 Cache integrated onto the processor die. It had the following basic characteristics:

- Using the Xeon cartridge format, it installed into the Slot 2 connector on the system board.
- The integrated L2 Cache was a unified, MESI, 4-Way set-associative cache available in 512KB and 1MB sizes.
- It had a full-speed BSB.
- It had a 133MHz FSB.
- The FSB arbitration scheme supported one or two processors in a rotational bus arbitration scheme.

6

Overview of the P6 Core

Description of the P6 Processor Core

Refer to chapter 5 in the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture, Second Edition* for a detailed description of the P6 family processor core.

Introduction to the Internal Architecture

Figure 6-1 on page 173 illustrates the basic elements that comprise a P6 family processor.

The FSB Interface Unit

The FSB interface Unit performs FSB transactions when requested to do so by the L2 Cache or the processor core. The processor's FSB Interface Unit also monitors the FSB and, when another FSB agent initiates a memory transaction, submits the memory address to the processor's internal caches to be snooped. The processor performs transactions on the FSB under the circumstances introduced in the following subsections.

IO Read or Write Transaction

An IO read or write transaction is initiated when the processor is executing an IO read or write instruction (IN, OUT, INS, or OUTS).

Memory Read Transaction

A memory read transaction is initiated under the following circumstances:

- When the processor core executes a load instruction.
- When the Instruction Prefetcher attempts to access a line of code that is not in the L1 Code Cache or the L2 Cache.
- When the processor is in Protected Mode and a new value is placed in one of the segment registers.
- When the processor is in Protected Mode and an interrupt or exception causes an entry to be read from the IDT.
- When the processor is in VM86 Mode and the VM86 task attempts to execute an IO read or write instruction, the processor performs a memory read from the IO Permission bit map in the task's TSS to determine if the task has permission to access the targeted IO port.
- When the processor is in VM86 Mode and the VM86 task attempts to execute a software interrupt instruction, the processor performs a memory read from the Interrupt Redirection bit map in the task's TSS to determine whether the interrupt should cause a GP exception and invoke the VMM, or should just be directed to the handler pointed to by the selected entry in the task's Real Mode IDT.
- When the processor is in Protected Mode with Paging enabled, the upper 20-bits of the linear memory address is submitted to the TLB cache for a lookup. Assuming that the targeted PTE is not present in the TLB, the processor must perform a memory read to obtain the selected PDE and another to obtain the selected PTE.

Memory Write Transaction

A memory write transaction is initiated under the following circumstances:

- When the processor core executes a store instruction to uncacheable memory space, the store is posted in the processor's Posted Write Buffer. All of the writes in this buffer are dumped to memory when a synchronizing event occurs (see "Forcing a Buffer Drain" on page 1083 in the book for more information).
- When the processor core has to evict a modified line from the L2 Cache to make room for a new line.

Chapter 6: Overview of the P6 Core

- When the processor core executes a WBINVD instruction, it performs a series of one or more memory writes to dump all modified lines to system memory.
- When one of the Pentium® III, Pentium® 4, or Pentium® M processor's WCBs becomes full, the processor automatically performs a memory write to write the line to memory.
- When one of the WCBs in the Pentium® Pro or the Pentium® II is full and another store to write combining (WC) memory is performed, the processor performs a memory write to write the line from the WCB to memory. This frees up the WCB to accept the currently executing store to WC memory.

There are actually two types of memory write transactions:

- **Memory Write.** This is the regular memory write transaction that is used for most memory writes. Intel®'s official name for this transaction type is the "Memory Write (may be retried)". The chipset is permitted to issue either a Deferred or Retry response to this transaction type.
- **Memory Line Writeback.** This transaction type is only used when the processor is casting a modified line from the L2 Cache back to system memory to make room for a new line being read into the cache. Intel®'s official name for this transaction type is the Memory Write (may not be retried). The chipset is not permitted to issue either a Deferred or Retry response to this transaction type. Rather, it must accept the memory write immediately so the processor can free up its Writeback Buffer to handle any subsequent castout of another modified line.

Memory Read and Invalidate Transaction

A Memory Read and Invalidate (MRI) transaction is performed under the following circumstances:

- When the processor executes a store that misses on the caches in a memory range designated as WB memory (cacheable Write Back memory; i.e., MESI memory). The processor performs a 32 byte MRI to obtain the line with the intention of immediately storing into it and marking it modified. This is also frequently referred to as a RFO (Read for Ownership) or as an RWITM (Read With Intent To Modify).
- When the processor executes a store that hits on a shared line (i.e., in the S state) in a memory range that is designated as WB memory, it performs an MRI for 0 bytes to kill the S copies of the line in the caches of other processors. After successful completion of the kill transaction, the processor completes the store into its cached copy of the line and marks the line Modified (M).

Special Transaction

The processor generates the Special transaction to broadcast a message to other agents in the system regarding an internal event within the processor. The event types (and the corresponding message types) are:

- NOP.
- Shutdown.
- Flush.
- Halt.
- Sync.
- Flush Acknowledge.
- Stop Clock Acknowledge.
- SMI Acknowledge.

See “The Special Transaction” on page 1306 in the book for more information.

Interrupt Acknowledge Transaction

When the processor’s Local APIC is disabled and an external hardware interrupt is delivered to the processor on its INTR input pin, the processor generates an Interrupt Acknowledge transaction on its FSB to request the interrupt vector from the external 8259A interrupt controller (assuming that the processor is enabled to recognize external hardware interrupts [i.e., EFlags[IF] = 1]).

Branch Trace Message Transaction

This feature is enabled if DebugCtlMSR[TR] = 1. Whenever the processor takes a branch (i.e., it jumps to the branch target address), the processor generates a BTM transaction on its FSB. The BTM transaction outputs the 32-bit linear memory address of the branch instruction as well as the 32-bit linear memory address that is being branched to. This permits a diagnostic tool attached to the FSB to track program execution.

Deferred Reply Transaction

This transaction type is never initiated by a processor. The FSB agent targeted by the processor in an earlier transaction may have responded to the transaction with a Deferred response. This indicates that the target has memorized the transaction as well as the ID of the agent that issued the transaction. When the device is ready to complete the transaction, it initiates a Deferred Reply transaction addressing the agent that originally issued the transaction request and completes the transaction.

The Backside Bus (BSB) Interface Unit

The BSB Unit interfaces the processor core to the unified L2 Cache.

The Unified L2 Cache

When a lookup in the L1 Data Cache or the L1 Code Cache results in a cache miss, that cache forwards the request to the L2 Cache for fulfillment. In the event of an L2 Cache miss, the cache forwards the request to the FSB Interface Unit to be fulfilled from system memory.

The L1 Data Cache

The L1 Data Cache services data load and store requests issued by the core's Load and Store execution units. When a miss occurs, the L1 Data Cache forwards the request to the L2 Cache for fulfillment (over the BSB).

The L1 Code Cache

The L1 Code Cache services instruction fetch requests issued by the instruction prefetcher. When a miss occurs, the L1 Code Cache forwards the request to the L2 Cache for fulfillment (over the BSB).

The Processor Core

The processor logic responsible for the following:

- Instruction fetch.
- Branch prediction.
- Parsing of the IA instruction stream to determine the boundaries between IA instructions.
- Decoding of IA instructions into fixed-length primitive instructions referred to as μ ops (micro-ops).
- Mapping accesses to the IA data register set to a larger physical register set.
- Dispatch, execution and retirement of μ ops.

The Local APIC Unit

The processor Local APIC is responsible for receiving interrupt requests from:

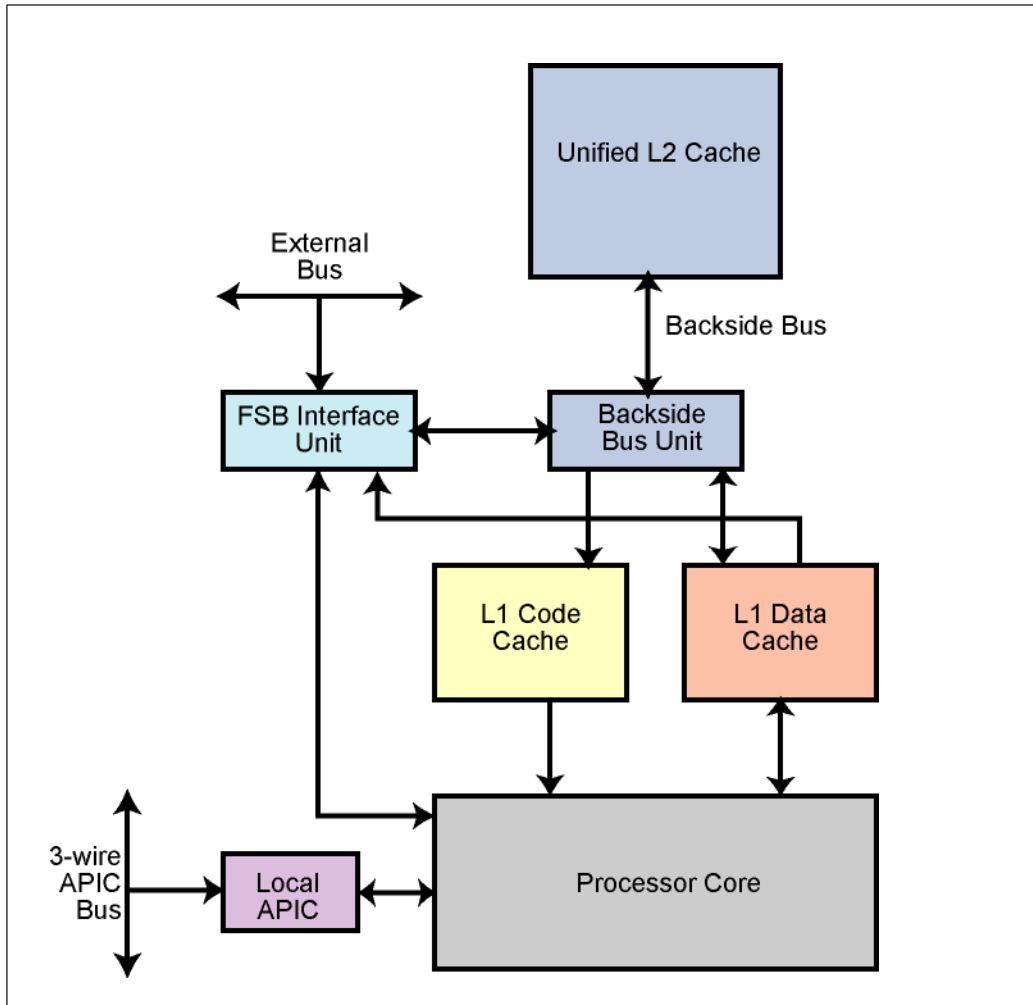
- Other processors.
- The processor's local interrupt input pins.
- The Local APIC's embedded timer.
- Local APIC error conditions.
- Performance monitoring logic.
- The IO APIC module.

These requests are then prioritized and forwarded to the processor core for execution.

A detailed description of Local APIC operation can be found in "The Local and IO APICs" on page 1497 in the book.

Chapter 6: Overview of the P6 Core

Figure 6-1: P6 Overview



The IA32 Instruction Set Is Variable Length

Legacy IA32 instructions are very complex in nature and vary in length. The minimum length is one byte and the maximum length is 15 bytes. As processor clock speeds have increased over the years, it became increasingly difficult to parse the IA32 instruction stream and to dispatch the instructions to the appropriate execution units at full speed.

IA32 Instructions Are Converted Into μ ops

The Pentium® Pro was the first x86 Intel® processor to translate IA instructions into fixed-length primitive instructions (μ ops) prior to dispatch and execution. All subsequent IA32 processors utilize this approach.

IA instructions are prefetched from memory and are placed into the unified L2 Cache and the L1 Code Cache. The L1 Code Cache feeds the instruction pipeline in the processor core. As blocks of IA instructions are obtained from the cache, the processor core parses them to identify the boundaries between instructions. It then decodes the variable-length IA instructions into μ ops. The μ ops are then placed in an instruction pool (referred to as the ReOrder Buffer, or ROB) where they await dispatch and execution. The dispatch logic can see all 40 entries in the ROB and can dispatch any instruction for execution as long as the appropriate execution unit and the data operands the instruction requires are available.

The processor:

1. Fetches the IA instructions from memory in strict program order.
2. Decodes, or translates, them (in strict program order) into one or more μ ops.
3. Places the μ ops into the ROB in strict program order.
4. Up to this point, the μ ops are kept in original program order. This part of the pipeline is known as the in-order front end. The processor then executes the μ ops in any order possible as the data and execution units required for each μ op become available. This is known as the out-of-order (OOO) portion of the pipeline.
5. Finally, the processor commits the results of the μ op execution to the processor's register set in the order of the original program flow. This is the retirement stage and is referred to as the in-order rear-end.

The IA32 Data Register Set

The Data Register Set Is Small

The IA register set only includes 16 registers that can be used to store data (see Figure 6-2 on page 177). They are the eight FP data registers (FP0 through FP7) and the EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP registers. The small number of IA data registers restricts the programmer to keeping a very small num-

Chapter 6: Overview of the P6 Core

ber of data variables close at hand in the registers. When these registers are all in use and the programmer needs to retrieve another data variable from memory, the programmer is forced to first copy the current contents of one of the IA registers back to memory before loading that register with new data from another area of memory. As an example, the following code fragment saves off the data in EAX before loading a new value into EAX:

```
mov  [mem1], eax
mov  eax, [mem2]
```

Due to the small data register set, this sequence is very common in x86 programs. Ordinarily, the processor could not execute these two instructions simultaneously (because the current contents of EAX must first be saved before the EAX register can be loaded with the new value). This is commonly referred to as a false register dependency that unnecessarily prevents the simultaneous execution of the two instructions.

However, the P6 processors are designed to use another, hidden register as the register to be written by the second instruction and can therefore execute both instructions simultaneously. In other words, at the same time that the first instruction is reading the current value from EAX so as to write it into memory, the second instruction can read from memory and write into a hidden register rather than the real EAX register. The value in the hidden copy of the EAX register can be copied into the actual EAX register at a later time.

Elimination of False Dependencies

As another example, consider the following code fragment:

```
mov  eax, 17
add  mem, eax
mov  eax, 3
add  eax, ebx
```

Ordinarily, these four instructions would have to be executed in serial fashion, one after the other, in order to produce the correct results. The P6 processors can execute the first and the third instructions simultaneously, moving the values 17h and 3h into two, separate hidden registers rather than into the actual EAX register. Likewise, the second and the fourth instructions can then be executed simultaneously.

Refer to Figure 6-3 on page 178. Assume that the instructions in the example code fragment will be placed into ROB entries 0 through 3. Prior to the placement of the μ ops in the ROB, they are tagged as follows:

- The first instruction will be placed in ROB entry 0. When the μ op is executed, its result (17h) will be placed in the Register field in ROB entry 0.
- The second μ op will be placed in ROB entry 1. Prior to placing it in the ROB, a tag is appended to it identifying the Register field of ROB entry 0 as the source of the EAX data (the register that will be loaded with the value 17h by the first instruction).
- The third μ op will be placed in ROB entry 2. When the μ op is executed, its result (3h) will be placed in the Register field in ROB entry 2.
- The fourth μ op will be placed in ROB entry 3. Prior to placing it in the ROB, a tag is appended to it identifying the Register field of ROB entry 2 as the source of the EAX data (the register that will be loaded with the value 3h by the third μ op).

The processor can then execute the first and third μ ops simultaneously, followed by the second and fourth μ ops simultaneously. This methodology eliminates many false dependencies between instructions, thereby permitting simultaneous execution and resulting in better performance. In the Register Alias Table (RAT) stage, the processor selects which of the 40 surrogate registers will actually be used by the μ op when it is executed.

Chapter 6: Overview of the P6 Core

Figure 6-2: IA32 Data Registers

Integer Data Registers				
	31	23	15	8 7 0
EAX			AH	AL
EBX			BH	BL
ECX			CH	CL
EDX			DH	DL
EBP			BP	
ESI			SI	
EDI			DI	
ESP			SP	

Floating Point Data Registers				
	79	78	64 63	0 Tag Field 1 0
R0	Sign	Exponent	Significand	
R1				
R2				
R3				
R4				
R5				
R6				
R7				

Chapter 6: Overview of the P6 Core

Table 6-1: Pipeline Stages

Stage	Description
IFU1	Instruction Fetch Unit stage 1. Load a 32-byte line from the code cache into the prefetch streaming buffer.
IFU2	Instruction Fetch Unit stage 2. Mark instruction boundaries. In the IFU2 stage, two operations are performed simultaneously: • The boundaries between IA32 instructions within a 16-byte block are identified. Intel® documents frequently refer to this as a line of information, but the term line is typically applied to the size of a block of information in a cache. To avoid possible confusion, the author therefore has decided not to refer to this as a line. • If any of the instructions within the 16-byte block are branches, the memory addresses that they were fetched from are presented to the BTB (Branch Target Buffer) for branch prediction. Refer to “The Branch Prediction Mechanisms” on page 181.
IFU3	Instruction Fetch Unit stage 3. Align the next three IA32 instructions for presentation to the appropriate decoders (see the DEC1 stage in Figure 6-5 on page 183) in the next stage.
DEC1	Decode stage 1. Translate (i.e., decode) the next three IA32 instructions into the corresponding μ op(s). Up to three IA32 instructions can be decoded simultaneously (if they are aligned with a decoder that can handle the instruction). In Figure 6-5 on page 183, the DEC1 stage illustrates the three decoders. The Complex Decoder can translate any IA instruction that translates into between one and four μ ops, while each of the Simple Decoders can translate any IA instruction that translates into a single μ op. If an IA instruction translates into more than four μ ops, it is submitted to the Micro-Instruction Sequencer (MIS) for decode. This is basically a μ op ROM containing the μ op sequence for each of the very complex IA instructions.
DEC2	Decode Stage 2. Place the μ ops in the Decoded Instruction Queue. Since some IA instructions (e.g., string operations) translate into rather large streams of μ ops and the Decoded Instruction Queue can only accept six μ ops per clock, it should be noted that this stage may have to be repeated a number of times (i.e, it may last for a number of processor clock ticks).

Table 6-1: Pipeline Stages (Continued)

Stage	Description
RAT	Register Alias Table and Allocator stage. The processor determines if the source operand(s) should be obtained from a real IA register or from an Alias Register entry in the ROB (see “Elimination of False Dependencies” on page 175). The latter case is true if a μ op previously placed in the ROB (from earlier in the code stream) will, when executed, produce the result required as a source operand by this μ op. In this case, the source field in this μ op is tagged to point to the respective Alias Register in the ROB.
ROB	ReOrder Buffer (ROB) stage. Move μ ops from the RAT / Allocator stage to the next sequential three ROB entries at the rate of three per clock. If all of the data required for execution of a μ op is available and an entry is available in the RS (Reservation Station) queue, also pass a copy of the μ op to the RS. The μ op then awaits the availability of the appropriate execution unit.
DIS	Dispatch stage. If not already done in the previous stage (because all of the data required by the μ op was not available or an entry wasn’t available in the RS), copy the μ op from the ROB to the RS, and then dispatch it to the appropriate execution unit for execution.
EX	Execution stage. Execute μ op. The number of clocks that a μ op takes to complete execution is μ op-dependent. Most μ ops can be executed in one clock.
RET1	Retirement stage 1. When a μ op in the ROB has completed execution, all conditional branches earlier in the code stream have been resolved and it is determined that the μ op should have been executed, it is marked as ready for retirement.
RET2	Retirement stage 2. Retire instruction. When the previous IA instruction has been retired and all of the μ ops associated with the next IA instruction have completed execution, the real IA register(s) affected by the IA instruction’s execution are updated with the instruction’s execution result (this is also referred to as “committing” the results to the machine state) and the μ op is deleted (i.e., retired) from ROB. Up to three μ ops are retired per clock in strict program order.

Instructions Are Executed Out of Order

The P6 processor family has an out of order execution engine. Once μ ops are placed in the ROB, they become visible to the μ op dispatch logic. The dispatch logic will dispatch an μ op for execution when all of the μ op's operands are available and the appropriate execution unit is available. The core can execute μ ops out of order. It can execute μ ops that programmatically fall after a conditional branch that has yet to be executed. In other words, μ ops can be speculatively executed. If it is later discovered that one or more μ ops that lie beyond a conditional branch should not have been executed, their results are not committed to the actual processor register set. Rather, they are discarded.

The Branch Prediction Mechanisms

486 Branch Handling

The 486 processor did not implement any form of branch prediction. The instruction decode logic didn't differentiate branches from other instruction types and therefore didn't alter prefetching. In other words, instruction prefetching always continued past a branch to the next sequential instruction. If the branch was taken when it was executed, the instructions in the prefetch queue and those in the earlier pipeline stages were flushed. This created a relatively minor bubble in the instruction pipeline (minor because the 486 pipeline had so few stages).

Pentium® Branch Prediction

The Pentium® processor implemented dynamic branch prediction using a very simple mechanism that yielded a typical 85% hit rate. As each prefetched instruction was passed into the Pentium®'s two instruction pipelines, the memory address it was fetched from was used to perform a lookup in the BTB (Branch Target Buffer). The BTB was implemented as a high-speed, look-aside cache. If there's a branch and it misses the BTB, it is predicted as not taken and the prefetch path is not altered. If it hits in the BTB, the state of the BTB entry's history bits are used to determine whether the branch is predicted as taken or not taken. When the branch moves further down the pipeline to the execution stage and is executed, its results (whether it is taken or not and, if taken, to what branch target address) are used to update the BTB. If the branch was incorrectly predicted, the μ ops in the two pipelines and those in the currently-active prefetch queue must be flushed. This didn't cause a terrible performance hit

because the Pentium® had relatively shallow instruction pipelines. For a more detailed discussion of the Pentium®'s branch prediction algorithm, refer to the MindShare book entitled *Pentium® Processor System Architecture* (published by Addison-Wesley).

P6 Branch Prediction

Mispredicted Branches are VERY Costly!

The P6 processor's instruction pipeline was deep (i.e., it consisted of many stages). Close to the beginning of the pipeline (in the IFU2 stage; see Figure 6-5 on page 183), the address of a branch instruction was submitted to the dynamic branch prediction logic (i.e., the BTB) for a lookup and, if it resulted in a hit, a prediction was made on whether or not the branch would be taken when the branch was finally executed by the JEU (Jump Execution Unit). The BTB only made predictions on branches that it had seen previously and for which it therefore had some execution history. Based on this prediction, the branch prediction logic took one of two actions:

- If the branch was predicted taken, the instructions that were fetched from memory locations along the fall-through path of execution (i.e., the instructions that follow immediately after the branch) were flushed from the 16-byte block of code that was currently in the IFU2 stage. The instructions currently in the prefetch streaming buffer were also flushed. The branch prediction logic provided the branch target address to the IFU1 stage and the prefetcher began to refill the streaming buffer with instructions from the predicted path.
- If the branch was predicted as not taken, the branch prediction logic did not flush the instructions that came after the branch in the 16-byte code block currently in the IFU2 stage. It also did not flush the streaming buffer and the prefetcher continued fetching code along the fall-through path.

The branch migrated through the pre-ROB pipeline stages, was placed in the ROB and, ultimately, was executed by the JEU (Jump Execution Unit). The series of μ ops from the predicted path following along behind it in the pipeline and were also placed in the ROB. Due to the processor's out-of-order execution engine, by the time the branch was executed, a number of the μ ops that fall after the branch in the program flow may have already completed execution.

When the branch was finally executed, all was well if the prediction was correct. However, if the prediction was incorrect:

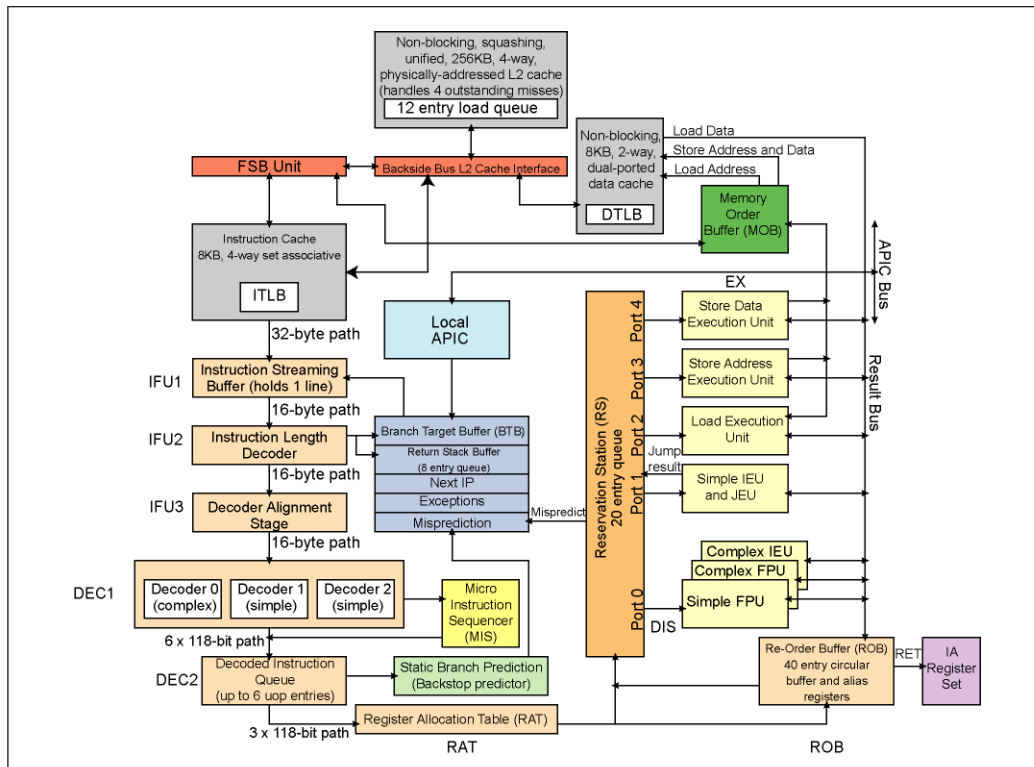
- all μ ops currently in the ROB that came after the branch were flushed from the ROB (along with their execution results if any of them had already been executed).

Chapter 6: Overview of the P6 Core

- all of the μ ops that fall after the branch that were currently in the RS or that were currently being executed had to be flushed.
- all μ ops in the earlier pipeline stages had to be flushed.
- the instruction streaming buffer had to be flushed.
- the prefetcher had to then issue a request for code starting at the correct address. This was supplied by the BTB which always kept a record of the branch target address as well as the fall through address.

Because of the large performance hit that resulted from a mispredicted branch, the designers of the processor incorporated three forms of branch prediction (Dynamic and Static Branch Prediction, as well as a Return Stack Buffer), and implemented a more robust Dynamic Branch Prediction algorithm than that used in the Pentium® processor. The sections that follow describe the Static and Dynamic Branch Prediction mechanisms.

Figure 6-5: A Branch Mispredict Is VERY Costly! Causes Entire Pipeline to Be Flushed



Dynamic Branch Prediction

General. Refer to Figure 6-6 on page 185. The P6 processor's BTB implemented a two-level, adaptive dynamic branch prediction algorithm. *Please note that the following initial discussion temporarily ignores the static branch prediction mechanism.* In the IFU2 pipeline stage, the address of any conditional branch instruction (up to four simultaneously) that resided within the 16-byte code block was submitted to the BTB for a lookup (in other words, the BTB could make up to four branch predictions simultaneously). The first time that a branch was seen by the BTB, it resulted in a BTB miss (the BTB had no history on how it would execute). When the branch finally arrived at the JEU (Jump Execution Unit) and was executed, an entry was made in the BTB. The entry recorded:

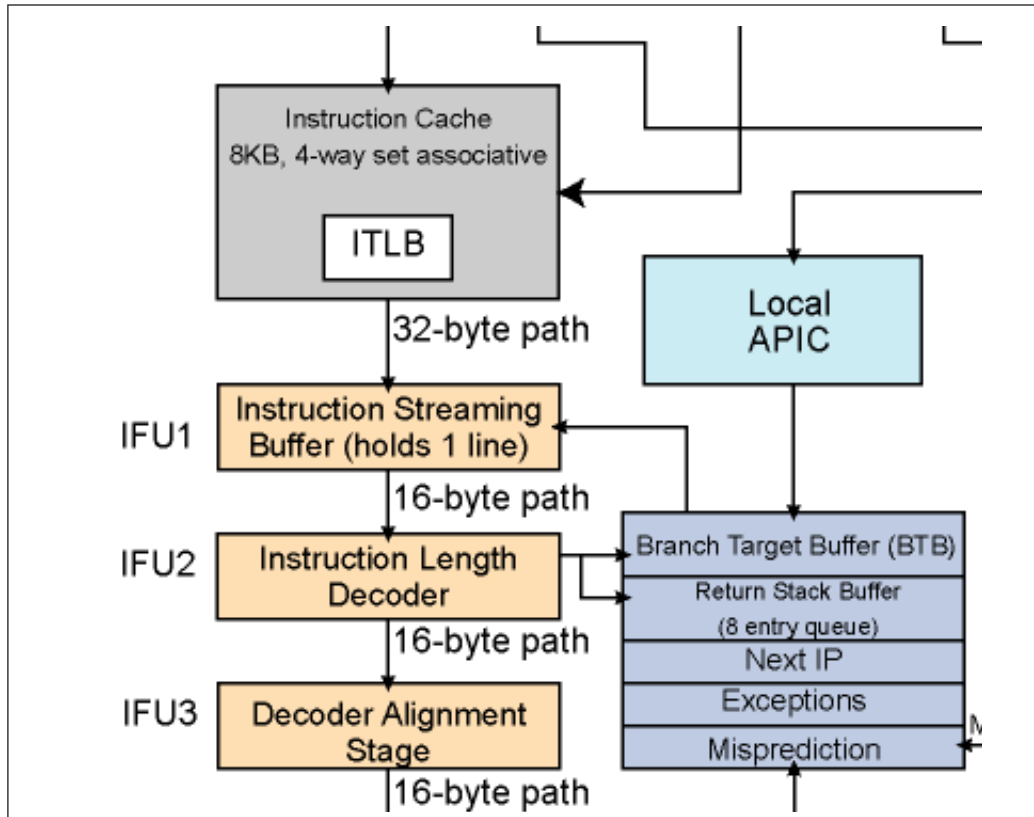
- the address that the branch was fetched from.
- whether or not the branch was taken and, if so, the branch target address.

The size and organization of the BTB is processor design-specific. The P6 implementations had a 4-way set-associate BTB with a total of 512 entries (double the Pentium® BTB's size).

Yeh's Prediction Algorithm. Unlike the Pentium®'s BTB, which used a simple counting mechanism (a variant on the Smith algorithm: a 2-bit counter that increments each time the branch is taken and decrements each time it's not; either of the higher two values indicate a taken prediction, while the lower two values indicate a not-taken prediction), the P6 processor's BTB was capable of recognizing behavior patterns such as taken/taken/not-taken. The algorithm used was referred to as Yeh's algorithm and was a two-level, adaptive algorithm. Each BTB entry used 4-bits to maintain history on the branch's behavior the last four times that it was executed. There are many ways in which this algorithm could have been implemented, but Intel® declined to describe the P6's implementation. For additional information on Yeh's algorithm, refer to a very good article in the 3/27/95 issue of Processor Report (Volume 9, Number 4). It is estimated that this algorithm can achieve accuracy of approximately 90-95% on SPECINT92 and better on SPECfp92.

Chapter 6: Overview of the P6 Core

Figure 6-6: Branch Target Buffer (BTB)



Return Stack Buffer (RSB)

The programmer calls a subroutine by executing a CALL instruction, explicitly citing the address to jump to. In other words, a CALL instruction is an unconditional branch and is therefore always correctly predicted as taken. When the CALL is executed, the address of the instruction that immediately follows the CALL is automatically pushed onto the stack by the processor. The programmer always ends the called routine with a RET (Return) instruction. The RET instruction pops the previously-pushed address off the stack and jumps to the instruction it points to in order to resume execution of the program that called the routine. In other words, the RET instruction is also an unconditional branch and can always be predicted as taken. While this is true, it doesn't address the question of where it will branch to.

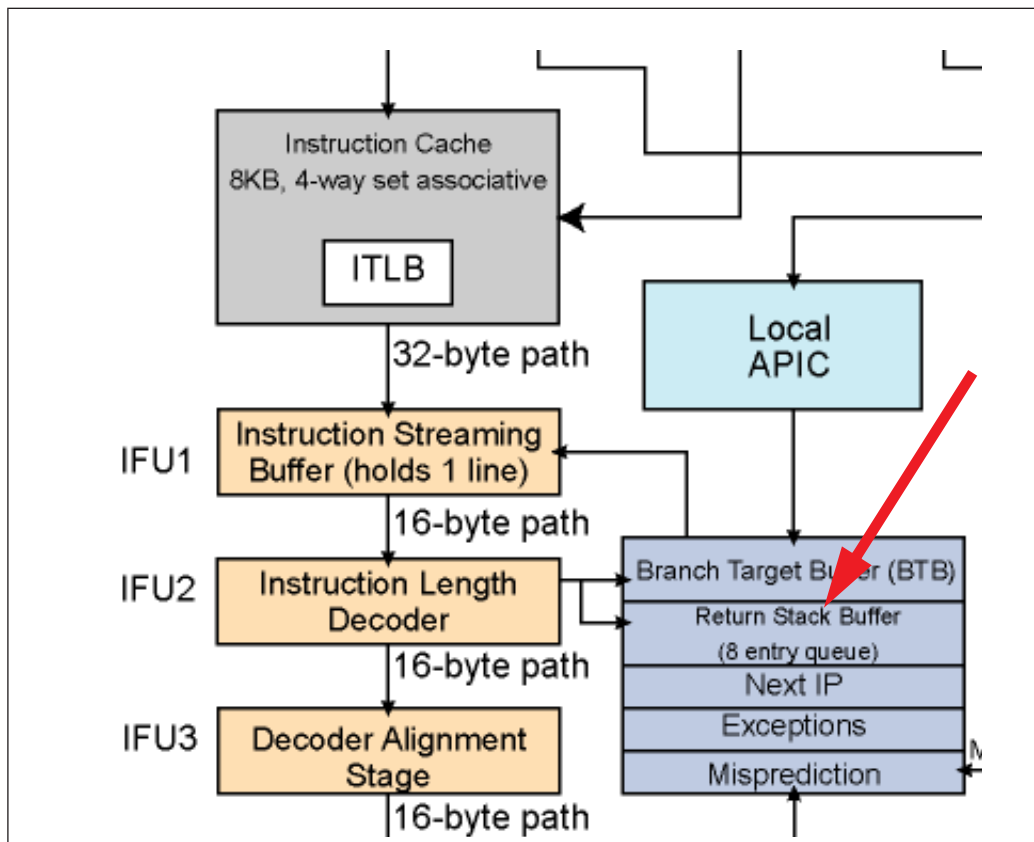
The Unabridged Pentium® 4

x86 processor implementations prior to the advent of the Pentium® Pro kept no record of the address pushed onto the stack and although it could be predicted that the branch would be taken, it could not “remember” where the branch would be taken to.

When a CALL instruction was executed, the return address was pushed into stack memory and was also pushed onto the RSB (see Figure 6-7 on page 186). The next RET instruction subsequently seen by the IFU2 stage caused the processor to access the top (i.e., most-recent) entry in the RSB. The branch prediction logic predicts a branch to the return address recorded in the selected RSB entry. In the event that the called routine alters the return address entry in stack memory, this will result in a misprediction.

As implemented in the P6 processors, the RSB had eight entries.

Figure 6-7: Return Stack Buffer (RSB)



Static Branch Prediction

The Static Branch Predictor was also referred to as the Backstop mechanism because it provides a backup to the Dynamic Branch Prediction logic (i.e., the BTB). Branches were submitted to the Static Branch Predictor logic in the DEC2 stage (see Figure 6-9 on page 188). The Static Branch Predictor decision tree is illustrated in Figure 6-8 on page 187.

For the most part, the Static Branch Predictor handled branches that missed the BTB. However, in the case of an unconditional IP-relative branch, the Static Branch Predictor always forced the prediction to the taken state (and updated the BTB if there was a disagreement).

Figure 6-8: Static Branch Prediction Algorithm

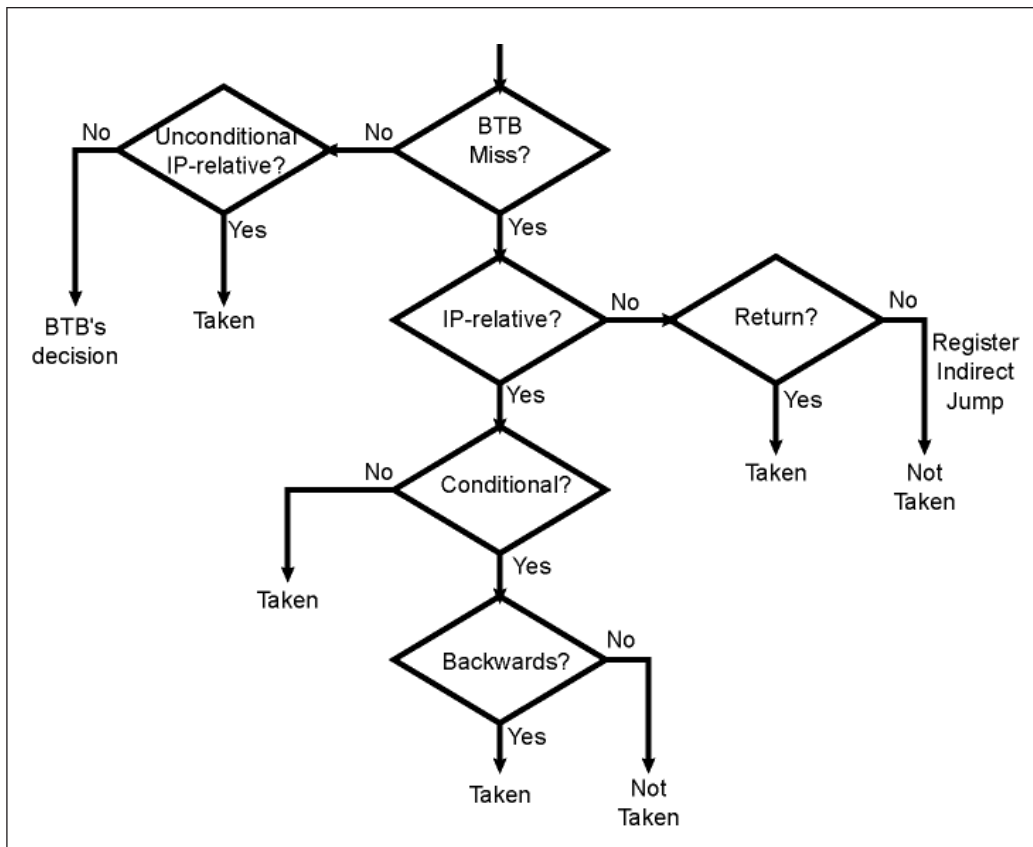
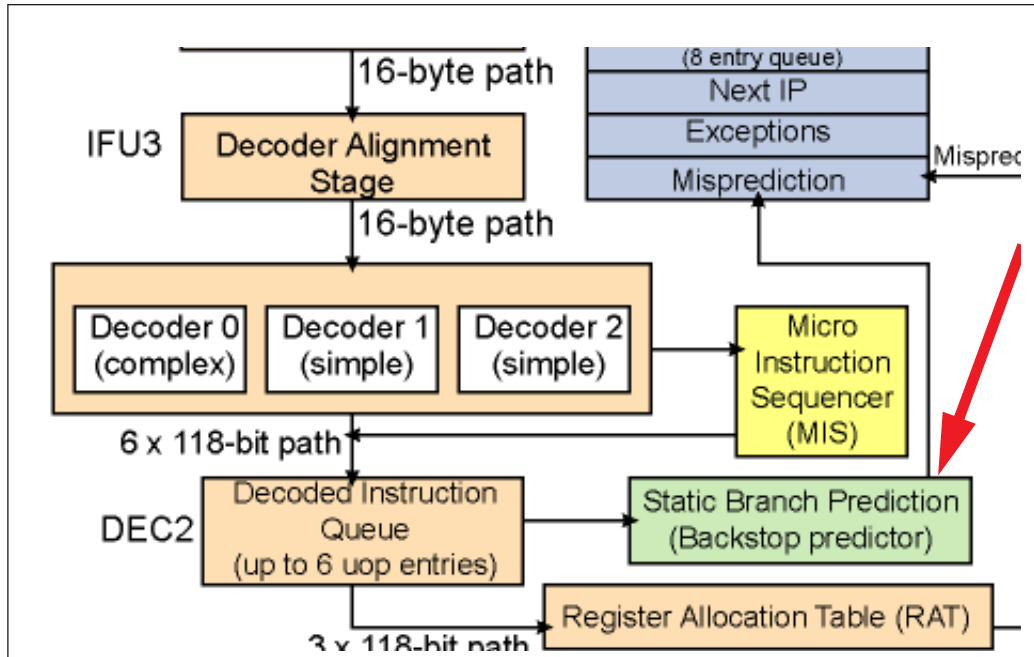


Figure 6-9: Static Branch Prediction Logic



Code Optimizations

General

This section highlights some of the code optimizations recommended by Intel® to yield the best P6 processor performance. It should be noted that all of these optimizations (with the exception of the last one) also apply to the Pentium® 4 and Pentium® M processors.

Reduce the Number of Branches

As described in “The Branch Prediction Mechanisms” on page 181, mispredicted branches cause a severe performance penalty. Wherever possible, eliminate branches. One excellent way to do this is by using the CMOV and FCMOV instructions [see “Conditional Move (CMOV) Eliminates Branches” on page 627 and “Conditional FP Move (FCMOV) Eliminates Branches” on page 627 in the book].

Follow the Static Branch Prediction Algorithm

In order to optimize performance for branches that miss the BTB, craft branches so as to take advantage of the Static Branch Prediction mechanism (see “Static Branch Prediction” on page 187).

Identify and Improve Unpredictable Branches

Indirect branches, such as switch statements, computed GOTOs, or CALLs through pointers can branch to an arbitrary number of target addresses. When the branch goes to a specific target address a large amount of the time, the Dynamic Branch Predictor will handle it quite well. However, if the target address is fairly random, the branch prediction logic may not have a very good rate of success. It would be better to replace the switch, etc., with a set of conditional branches that the branch prediction logic can handle well.

Don't Intermingle Code and Data

Don't intermingle code and data within the same cache line. The line can then end up residing in both the code and data caches. If the data is modified, the processor treats this as if it is Self-Modifying Code (SMC; see “Self-Modifying Code and the P6” on page 281), resulting in poor performance.

Align Data

Multi-byte data objects should be placed in memory aligned within a dword wherever possible. A data object (consisting of 2 or 4 bytes) that straddles a dword boundary may cause a lengthy delay to read or write the object. The worst-case scenario would be a data object that straddles a page address boundary (i.e., a 4KB page) and causes two Page Fault exceptions (because neither page is currently in memory). This would result in a stall of the current program until both pages had been read from mass storage into memory.

Avoid Serializing Instructions

See “Serializing Instructions” on page 1079 in the book. Serializing instructions such as CPUID restrain the processor from performing speculative code execution and have a serious impact on program execution. While they must be used sometimes to achieve a specific goal, they should be used as sparingly as possible. The following instructions cause execution serialization on the P6 processor family:

- Privileged instructions—MOV to control register, MOV to debug register, WRMSR, INVD, INVLPG, WBINVD, LGDT, LLDT, LIDT, AND LTR.
- Non-privileged instructions—CUID, IRET, and RSM.

Where Possible, Do Context Switches in Software

When the processor hardware is used to perform a task switch (by selecting a Task Gate descriptor), the bulk of the processor's register set is automatically saved to the TSS (Task State Segment) for the program being suspended, and the register set is then reloaded with the register image from the TSS associated with the program being started (or resumed). This save and reload takes quite a bit of time. In addition, the processor dumps all of the PTEs currently cached in its TLBs when a task switch occurs. This causes performance degradation in the task being started or resumed (because the TLB is empty). See "386 Demand Mode Paging" on page 209 in the book for a complete description of Paging.

It may be possible for software to suspend the current task by saving a smaller subset of the register set (rather than the entire register set), and it may be possible to start (or resume) the new task by loading just a few of the registers. In this case, performance would be aided if the programmer performs the state save and reload using software.

Eliminate Partial Stalls: Small Write Followed by Full-Register Read

If a full 32-bit register (e.g., EAX, EBX, ECX, or EDX) is read (e.g., MOV EBX, EAX) after part of the register is written to (e.g., MOV AL, 5), the P6 processor experiences a stall of at least 7 clocks (and maybe much longer). The μ op that reads from the full register is stalled until the partial writes are retired, writing the data to the appropriate portions of the real IA register. Only then is the value read from the full, 32-bit real IA register. In addition, none of the μ ops that follow the full register read μ op will be executed until the full register read completes.

Since 16-bit code performs partial register writes a lot, the P6 processor suffers poor performance when executing 16-bit code.

Data Segment Register Changes Serialize Execution

16-bit code frequently changes the contents of data segment registers (DS, ES, FS, and GS). (In the Pentium® Pro processor) Unlike the processor's GPRs, data segment register references by a μ op were not tagged to reference Alias Registers in the ROB. A write to a data segment register (DS, ES, FS or GS) immediately changed the value in the real, IA data segment register. μ ops that resided downstream from the μ op that changed the register may perform accesses (loads or stores) within the affected data segment. If the Pentium® Pro processor were permitted to speculatively execute instructions beyond the one that changed the data segment register before the μ op that altered the segment register had completed execution, they would be using the old, stale contents of the segment register and would address the wrong location.

For this reason, the Pentium® Pro processor would not execute any instructions beyond the segment register load until the load had completed. The processor's performance degraded because it was restrained from out-of-order execution. *Since 16-bit code changes the contents of the data segment registers a lot, the Pentium® Pro processor suffered poor performance when executing 16-bit code.*

Intel® fixed this problem in the Pentium® II and subsequent IA32 processors by aliasing the data segment registers to Alias Register entries in the ROB (as was already done for the GPRs).

BSP Selection Process

The Boot Strap Processor (BSP) is the processor which starts up when the system is powered up (or when the system is reset). It is the processor tasked with:

- Running the POST (Power On Self Test) code from the BIOS ROM.
- Performing a preliminary system configuration.
- Detecting the presence of any other processors.
- Causing the other processors (if there are any) to execute a program one at a time. The execution of this program by each of the other processors causes each of the processors to be configured, a CPU counter memory location to be incremented, an entry to be created in the Multiprocessing (MP) Table, and an entry to be created in the ACPI Table.

In a multiprocessor P6 system, the manner in which the BSP was selected was rather laborious. On the removal of reset, the following automatic procedure would be performed by the processors:

- The Local APIC in each of the processors would begin arbitrating for ownership of the 3-wire APIC bus.
- The winning processor's Local APIC would transmit a BIPI (Bootstrap Inter Processor Interrupt) message to the other processors.
- The winning processor sets the BSP bit in its APIC_BASE MSR to mark itself as the BSP.
- Each of the losing processors' Local APICs receives the message and clears the BSP bit in its respective APIC_BASE MSR. By doing so, they each mark themselves as APs (Application Processors—losers, in other words).
- Each of the other Local APICs would, in turn, acquire ownership of the APIC bus and transmit its BIPI message (but it's a moot point because the winner has already been selected).
- The BSP then regains ownership of the APIC bus and transmits the FIPI (Final Inter Processor Interrupt) message to the chipset to announce itself as the BSP. The chipset could be designed to latch and remember which of the processors acted as the BSP for this power up session. In the event of a problem, this could be used as diagnostic information later on.
- The BSP then arbitrates for ownership of the FSB to start fetching the BIOS code from the BIOS ROM.
- The APs stay halted and await the receipt of a SIPI (Startup Inter Processor Interrupt) message if and when a multiprocessor capable OS is booted.

7

Overview of the P6 FSB Basics

Note

It should be noted that the Pentium® 4 and Pentium® M FSB implementation uses the same principles introduced in this section.

Agents

Agent Types

All devices that reside on the FSB are referred to as *agents*. Basically, there are three type of agents:

- The **Request Agent** is the device that initiates a transaction by issuing a request (e.g., a memory read or write, or an IO read or write). It is commonly referred to as the transaction *initiator*.
- The **Response Agent** is the device on the FSB that acts as the target of the transaction (e.g., an IO target or a memory target).
- The **Snoop Agents** are any devices on the FSB that have memory caches (usually processors, in addition to the processors there could be an external cache that resides on the bus). Whenever any Request Agent starts a transaction, the request is latched by all FSB agents including the snoopers. If it is

a memory transaction, the memory address is then submitted to the snoopers' caches for a lookup and the results of this "snoop" are reported back to the Request Agent. The results will be one of the following:

- **Snoop miss**—None of the snoopers has a copy of the addressed line at all.
- **Snoop hit**—One or more of the snoopers has a copy of the addressed line in the E (Exclusive) or S (Shared) state and it hasn't been updated since being read from memory.
- **Snoop hit on a modified line**—One of the snoopers has a copy of the line and one or more of the bytes in the copy were updated since the line was copied into its cache from memory. The line in memory is stale.

Multiple Personalities

Some agents are only capable of acting as Response Agents (i.e., as the target of a transaction). As an example, the system memory controller typically acts as the target of memory reads and writes. It never initiates transactions, nor does it ever act as a Snoop Agent in a transaction.

Some agents are capable of acting as the Response Agent in some transactions and as the Request Agent for other transactions. As an example, in Figure 7-1 on page 195 the North Bridge (i.e., the FSB-to-PCI bridge) may:

- act as the Response Agent (i.e., the target) of a processor-initiated transaction to read data from an IO port in a PCI device beyond the bridge.
- act as the request initiator of a snoop transaction when a PCI master is accessing system memory.

An agent may act as the Request Agent for transactions that it initiates and as the Snoop Agent for memory transactions initiated by others. An example would be a processor. It not only initiates transactions on an as-needed basis, but also snoops memory transactions that are initiated by the other processors or by the FSB/PCI bridge (for PCI masters).

Uniprocessor vs. Multiprocessor Bus

The Pentium® FSB was ill-suited in a platform wherein multiple processors resided on the FSB (see Figure 7-1 on page 195). Each of the processors would have to compete for ownership of the FSB if they needed to perform a FSB transaction.

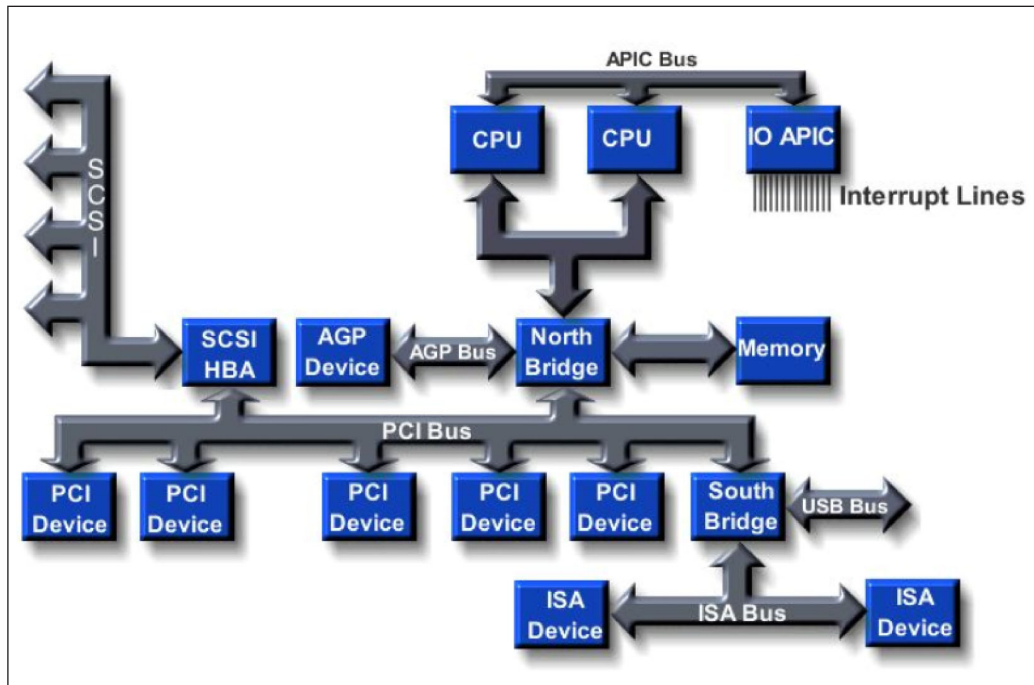
Chapter 7: Overview of the P6 FSB Basics

The P6 FSB was specifically designed to support multiple processors on the same FSB. The following major changes were made:

- In a typical P6 FSB environment, up to eight transactions could be in progress simultaneously at various stages of completion.
- If the target of a transaction could not deal with a new transaction at a given moment in time (e.g., due to a logic busy condition), rather than tying up the FSB by inserting wait states in the transaction, it would issue a *Retry* response to the Request Agent. This caused the initiator to wait a while and try the transaction again at a later time. This freed up the FSB to be used by other Request Agents.
- If the target of a read or write transaction realized that it would take a fairly long time to complete a data transfer (i.e., provide read data or accept write data), it could issue a *Deferred Response* to instruct the Request Agent to break the connection. The device that issued the *Deferred Response* would initiate a *Deferred Reply* transaction to complete the transfer later. This is referred to as *transaction deferral*.

These mechanisms prevent any properly-designed FSB agent from tying up the FSB for extended periods of time.

Figure 7-1: Block Diagram of a Typical System



Request Agents

The Request Agent Types

There are two types of Request Agents:

- **Symmetric Request Agents.** Most typically, these are the processors. With regard to FSB arbitration, the symmetric Request Agents have equal importance with respect to each other and use a rotational priority scheme for FSB arbitration. Note that a custom-designed Request Agent other than a processor could be designed to operate as a symmetric agent. The symmetric agent FSB arbitration scheme supports up to but no more than four symmetric Request Agents in the rotation.
- **Priority Request Agents.** The system designer may include one or more Request Agents other than the symmetric Request Agents. If a Priority Agent is competing against the symmetric agents for bus ownership, it wins and they lose (with one exception that is covered later).

The Agent ID

What the Agent ID Is Used For

When a Request Agent issues a transaction request, two of the items of information that it provides to the addressed Response Agent are:

- The Request Agent's unique agent ID.
- A unique transaction ID.

This information is only used by the Response Agent if it chooses to memorize the transaction and break the connection with the Request Agent (by issuing a Deferred Response). The Response Agent then processes the read or write request off-line and, when the data has been read or written, it arbitrates for the FSB and issues a *Deferred Reply transaction*. Using the agent and transaction IDs delivered to it in the original transaction request, it addresses the Request Agent that initially issued the read or write request and completes the transaction. A detailed description of deferred transactions can be found in "Pentium® 4 FSB Transaction Deferral" on page 1277 in the book.

Chapter 7: Overview of the P6 FSB Basics

How the Agent ID Is Assigned

Each agent that is capable of initiating transaction requests must be assigned a unique agent ID at startup time. The agent typically accomplishes this by sampling one or more of its inputs on the trailing-edge of reset or when the power supply output voltages stabilize (i.e., on the rising-edge of the POWERGOOD signal). The manner in which the processors obtain their agent IDs is covered in “Assignment of IDs to the Processor” on page 860 in the book.

The Transaction Phases

The Pentium® Transaction Phases

Each transaction initiated on the Pentium® FSB passed through three phases (or stages) from its inception to its completion:

- **Arbitration Phase**—If the initiator didn’t currently own the FSB, it requested ownership from the arbiter and awaited the granting of ownership.
- **Address phase**—After acquiring FSB ownership, the initiator drove out an address and transaction type and asserted ADS# (Address Strobe) for one clock to indicate that a transaction had been initiated and that a valid address and transaction type had been presented on the FSB. All agents latched the information and began the decode to determine which of them was the target of the transaction.
- **Data Phase**— The initiator waited for the target to provide the requested read data or to accept the write data.

The P6 FSB Transaction Phases

Each transaction initiated on the P6 FSB proceeded through the following phases from its inception to its completion (note that some transaction types did not require a data transfer and therefore did not include the Data Phase):

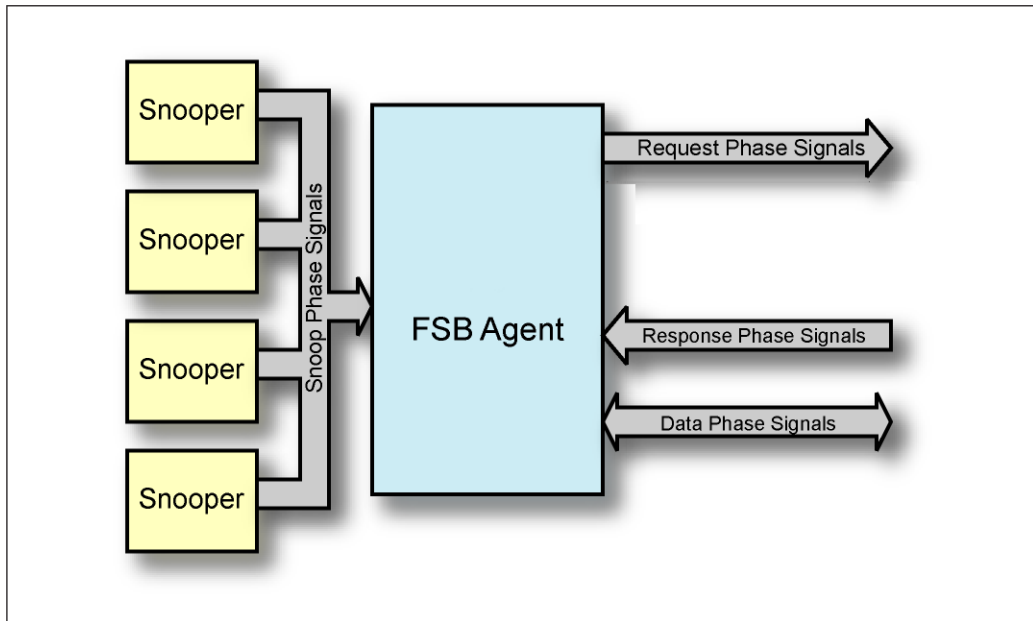
- Arbitration Phase.
- Request Phase.
- Error Phase.
- Snoop Phase.
- Response Phase.
- Data Phase.

Transaction Pipelining

The FSB Is Divided into Signal Groups

Refer to Figure 7-2 on page 198. The FSB is divided into signal groups. Each signal group is only used during a particular phase of the transaction.

Figure 7-2: Bus Signal Groups



Step 1: Gain Ownership of the Request Phase Signal Group

The FSB is divided into groups of signals, each of which is only used during a particular phase of a transaction. When a Request Agent wishes to acquire FSB ownership in order to issue a transaction request, it arbitrates for ownership of the portion of the FSB referred to as the *Request Phase signal group*.

Step 2: Issue the Transaction Request

The transaction request is driven onto the Request Phase signal group during the Request Phase of the transaction. All FSB agents are required to latch the new transaction request. Once the Request Agent has completed delivery of the transaction request to the other FSB agents, it ceases to drive the signals that comprise the Request Phase signal group. One BCLK period after the current Request Phase signal group owner begins to turn off its drivers, the same agent or another agent could take ownership of the Request Phase signal group and begin to drive out another transaction request.

Step 3: Yield Request Phase Signal Group, Proceed to Next Group

The Request Agent doesn't request ownership of the signal group used during the phase that immediately follows the Request Phase. As the Request Agent finishes each phase, it relinquishes control of the signal group used in the current phase and automatically takes ownership (or observes, as the case may be) of the signal group used in the next phase.

Phases Proceed in Predefined Order

The phases are passed through in a predefined order:

1. **Request Phase.** A detailed description can be found in chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.
2. **Error Phase.** A detailed description can be found in chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition. *Please note that this phase was eliminated in the Pentium® 4/Pentium® M processors.*
3. **Snoop Phase.** A detailed description can be found in chapter 12 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.
4. **Response Phase.** A detailed description can be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.
5. **Data Phase.** A detailed description can be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

The sections that follow provide a brief description of the phases.

The Request Phase

The Request Agent uses the Request Phase signal group to issue the transaction request to the other FSB agents and all of them latch it. The Response Agents (i.e., the potential targets) begin the decode to determine which of them is the target of the transaction. The Snoop Agents determine if it's a memory transaction. If it is, they must perform a cache lookup and deliver the snoop result during the Snoop Phase of this transaction. A detailed description of the Request Phase can be found in chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

The Error Phase

Note: This phase was eliminated in the Pentium® 4/Pentium® M protocol. The other FSB agents check the parity of the request just latched. If the parity is correct, no action is taken. If one or more of them detect a parity error (in other words, the request was corrupted in flight), they each assert the AERR# signal for one BCLK at the end of the Error Phase (AERR# is one of the few FSB signals that is a multiple-owner, open-drain signal). The Request Agent checks the AERR# signal at the end of the Error Phase to determine if all agents received the request without error. If there was an error, the remaining phases of the transaction are cancelled (the Request Agent may choose to wait a few clocks and reattempt the transaction from scratch). If the request was received by all agents without error, the Request Agent proceeds from the Error Phase to the Snoop Phase of the transaction. A detailed description of the Error Phase can be found in chapter 12 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

The Snoop Phase

After checking the state of the AERR# signal (and assuming that it wasn't asserted), the Request Agent proceeds to the Snoop Phase and begins sampling the snoop result signal group (which consists of HIT# and HITM#). The Snoop Agents are responsible for delivering the result of their cache snoop during this phase. In addition, the currently-addressed Response Agent can assert the DEFER# signal if it intends to issue a Retry or a Deferred Response during the Response Phase. Once the snoop result has been received, the Request Agent stops sampling the snoop result signals and the Snoop Agents cease driving them. A detailed description of the Snoop Phase can be found in chapter 12 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Chapter 7: Overview of the P6 FSB Basics

The Response Phase

Having completed the Snoop Phase, the Request Agent proceeds to the Response Phase and begins sampling the response signal group (consisting of RS[2:0]#). The Response Agent (i.e., the target addressed by the request) is responsible for delivering its response (i.e., how it intends to handle the request) to the Request Agent during this phase. Once the response has been received, the Request Agent stops sampling the response signal group and the Response Agent ceases to drive the response. The response delivered indicates one of the following:

- It tells the Request Agent to re-arbitrate for ownership of the Request Phase signal group and **retry** the transaction again.
- It indicates a **hard failure** (not to be retried and there is no data transfer; typically results in a Machine Check exception).
- If it's a write, that the Response Agent **will accept** the **data** in the Data Phase.
- If it's a read, that the Response Agent **will supply** the **data** during the Data Phase.
- If a snoop indicated a hit on a modified line (HITM# was asserted), the entire line will be transferred from the snoop to system memory (and, if it's a read, to the Request Agent as well)—this is referred to as an **Implicit Writeback Response**.
- It instructs the Request Agent to end the transaction with no data transferred. The Response Agent subsequently obtains or delivers the data off-line and then initiates a Deferred Reply transaction to indicate completion. This is referred to as the **Deferred Response**.

A detailed description of the Response Phase can be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

The Data Phase(s)

Most transactions involve a data transfer, but some don't. If a transaction involves a data transfer, the Request Agent proceeds to the Data Phase(s) of the transaction. If it's a write transaction, the Request Agent takes ownership of the data bus and delivers the data to the Response Agent during this phase. If it's a read transaction, the Response Agent takes ownership of the data bus and delivers the data to the Request Agent. Once the Data Phase completes, the transaction has completed and the Request and Response Agents stop using the data bus. A detailed description of the Data Phase can be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Next Agent Can't Use Signal Group Until Current Agent Done With It

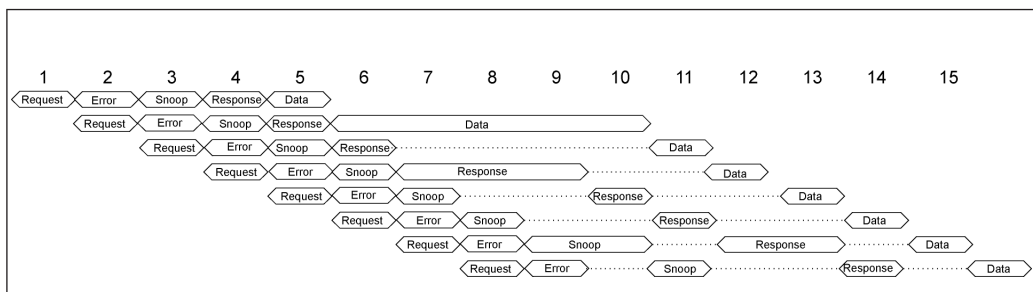
When a Request Agent has finished driving its transaction request onto the Request Phase signal group and has backed its output drivers from those signals, the same agent or a different Request Agent can take ownership of the Request Phase signal group and issue another transaction. Additional requests can be issued by the same or different Request Agents as the current Request Agent completes delivery of a request. See Figure 7-3 on page 202.

At a given instant, a number of transactions may be in progress at various stages of completion. As an example, at time 8 in Figure 7-3 on page 202 there are eight transactions active on the FSB at various phases on their way to completion. The figure also highlights that some of the phases can take some time complete:

- The **Snoop Phase** (see times 9 and 10 in the figure) can be stretched by the Snoopers if any of them needs more time in order to produce the results of the cache lookup.
- The **Response Phase** (see times 7, 8, and 9) can be stretched by the currently-addressed target if it requires extra time before it decides what its response to the transaction will be.
- The **Data Phase** (see times 6 through 10) of the transaction cannot begin until the Request and Response Agents are ready to transfer the data.

In those cases where the current owner(s) of a signal group is not yet finished using it, the next owner(s) cannot start using it until the previous agent's finished and the signal group has been completely released.

Figure 7-3: Pipelined Transactions



Transaction Tracking

Request Agent Transaction Tracking

It should be obvious that a Request Agent must keep track of each transaction that it has issued as each passes through the various phases on its way to completion. The Request Agent has distinct responsibilities during each phase:

- It issues the transaction request during the **Request Phase**.
- It checks that the request transmission completed without error during the **Error Phase**.
- It waits for and checks the snoop result during the **Snoop Phase**.
- It waits for and checks the response from the Response Agent during the **Response Phase**.
- During the **Data Phase** of a read or write transaction, it accepts or sources data.
- If, in the Response Phase, the Response Agent indicates that it wishes to **defer** completion until a later time, the Request Agent must remember this transaction (in its Deferred Transaction Queue) until the Response Agent initiates a Deferred Reply transaction at a later time.

In order to fulfill these responsibilities, the Request Agent must keep track of all outstanding transactions issued by itself as well as those issued by other Request Agents. As each of its outstanding transactions enters a new phase, it must interact with the appropriate signal group.

Snoop Agent Transaction Tracking

The group of Snoop Agents that reside on the FSB must also keep track of each transaction currently outstanding on the bus:

- At the appropriate time, they must present the snoop result for each transaction. As illustrated in Figure 7-3 on page 202 (in time 11), sometimes the delivery of the snoop result for a particular transaction must be delayed until all of the snoopers have presented the snoop result for the previous transaction.
- When a snoop results in a hit on a modified line (HITM# is asserted), the snoopers must be prepared to write back the modified line to the currently-addressed memory Response Agent (and to the Request Agent if it's a read) during the Data Phase of the transaction.

Response Agent Transaction Tracking

A Response Agent must track the following:

- It must latch each new transaction request and determine if it is the target of the transaction.
- If it is the target and it intends to issue a Retry or Deferred Response during the Response Phase, it must assert DEFER# during the Snoop Phase.
- If it is the target, the Response Agent must indicate (during the Response Phase of the transaction) how it wants to handle the transaction.
- If it is the target and this is a read, the Response Agent will have to return data to the Request Agent when the Data Phase is entered.
- If it is the target and this is a write, the Response Agent must be prepared to accept the write data.
- If it is the target and the snoop result is a hit on a modified line (HITM# was asserted), the Response Agent must be prepared to accept the line from the snoopers during the Data Phase of the transaction.

The In Order Queue (IOQ)

In order to properly interact with the FSB at each stage of the appropriate transactions, each FSB agent must maintain a record of all transactions currently in progress, what phase each is currently in, and what responsibilities (if any) it has during each phase. This record is kept in a buffer referred to as the agent's In Order Queue (IOQ). When each transaction receives a response guaranteeing that the transaction will be completed now (in other words, it isn't being deferred or it doesn't have to be retried), the transaction is deleted from the queue.

Pentium® 4/Pentium® M Differences

The Error Phase was eliminated in the Pentium® 4/Pentium® M protocol.

8

P6 FSB Electrical Overview

Applicability to the Pentium® 4/Pentium® M

Different Voltage Levels

The termination voltage (V_{tt}) and the reference voltage used on the Pentium® 4 are lower than those used on the P6 processor family.

AGTL+

Starting with the Pentium® II processor with a 100MHz FSB, the protocol for returning an asserted signal (i.e., electrically low) to the deasserted state (i.e., electrically high) was changed. Rather than allowing the termination resistors to pull a signal back high, the FSB agent that was driving a signal low must actively drive the signal high for one BCLK cycle. This returns the signal to the high state quickly and reduces ringing and settling time.

Introduction

One of the keys to a high-speed signaling environment is to utilize a low-voltage swing (LVS) to change the state of a signal from one state to the other. The P6 FSB fell into this category. It permitted the operation of the FSB at speeds of 60MHz or higher (a fast bus at the time). The FSB was implemented using a modified version of the industry standard GTL (Gunning Transceiver Logic) specification, referred to by Intel® as GTL+. The spec was modified to provide larger noise margins and reduce ringing. This was accomplished by using a higher termination voltage and controlling the edge rates. The net result was that the FSB supported more electrical loads (up to eight devices) than it would if implemented using the standard GTL spec. The sections that follow introduce the basic concepts behind the FSB's operation (from an electrical perspective). A detailed GTL+ spec can be found on the Intel® developers' website.

Everything's Relative

Every device residing on the FSB implements a comparator (i.e., a differential receiver) at each of its inputs (see Figure 8-1 on page 207). When a device samples the state of one of its FSB inputs (sampling occurs on the rising-edge of the Bus Clock—BCLK), the value sampled (i.e., the voltage level) is compared to a standard reference voltage supplied to the device by a voltage divider network on the system board (see Figure 8-2 on page 207), where V_{tt} is the pull-up, or termination voltage). $V_{tt} = 1.5V_{dc} \pm 10\%$ (see Figure 8-3 on page 208). If the sampled voltage is higher than the reference voltage (see Figure 8-5 on page 209) by a sufficient margin, it represents an electrical high, while if it's sufficiently lower than the reference voltage (see Figure 8-4 on page 208), it represents an electrical low (the margins are described in "How High is High and How Low is Low?" on page 212).

The differential receivers in FSB devices determine highs and lows in a voltage-independent manner. The devices don't have to be redesigned if different voltage levels are used to indicate a high and a low— just supply a different reference voltage to the device!

Chapter 8: P6 FSB Electrical Overview

Figure 8-1: A GTL+ Input Comparator

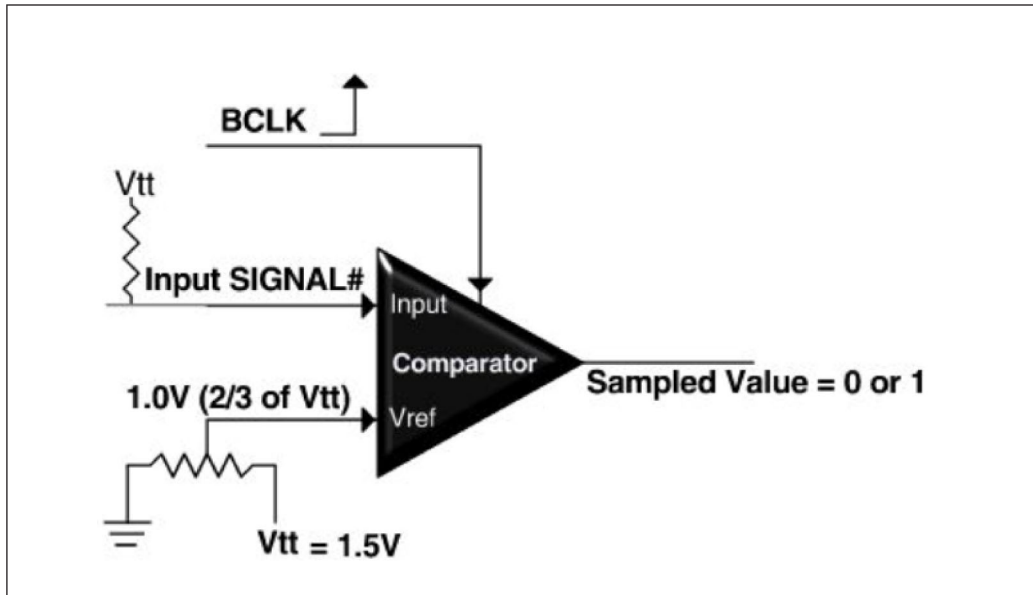
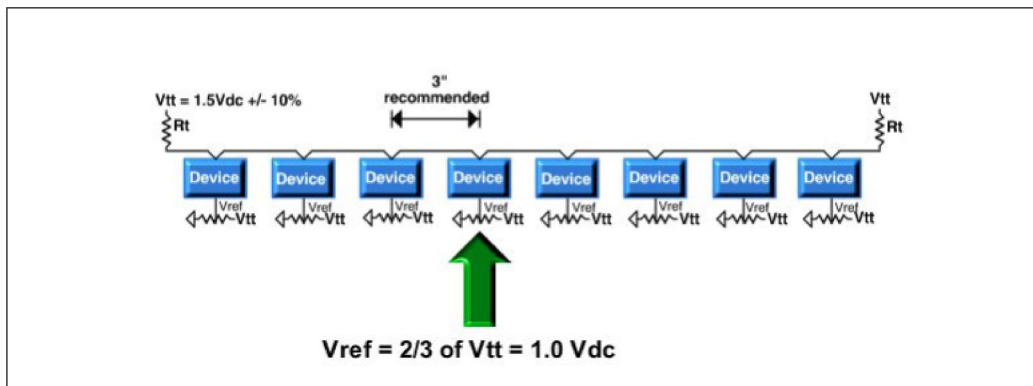


Figure 8-2: Voltage Divider Sources Reference Voltage (Vref) to Device's Input Comparators



The Unabridged Pentium® 4

Figure 8-3: The Pull-up Voltage (V_{tt}) = 1.5Vdc

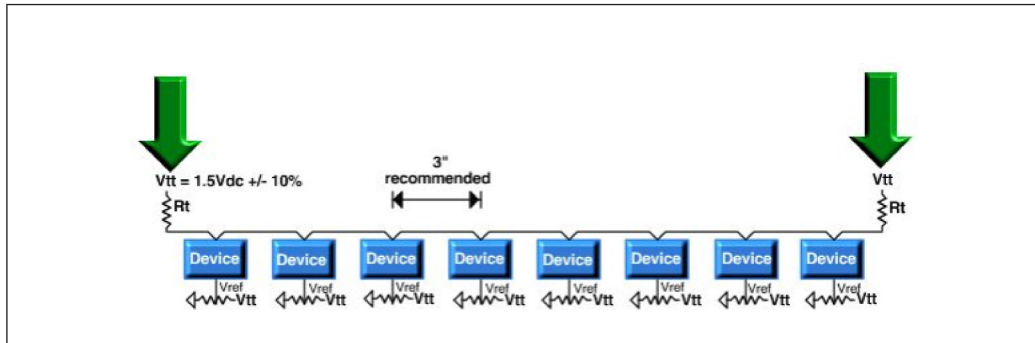
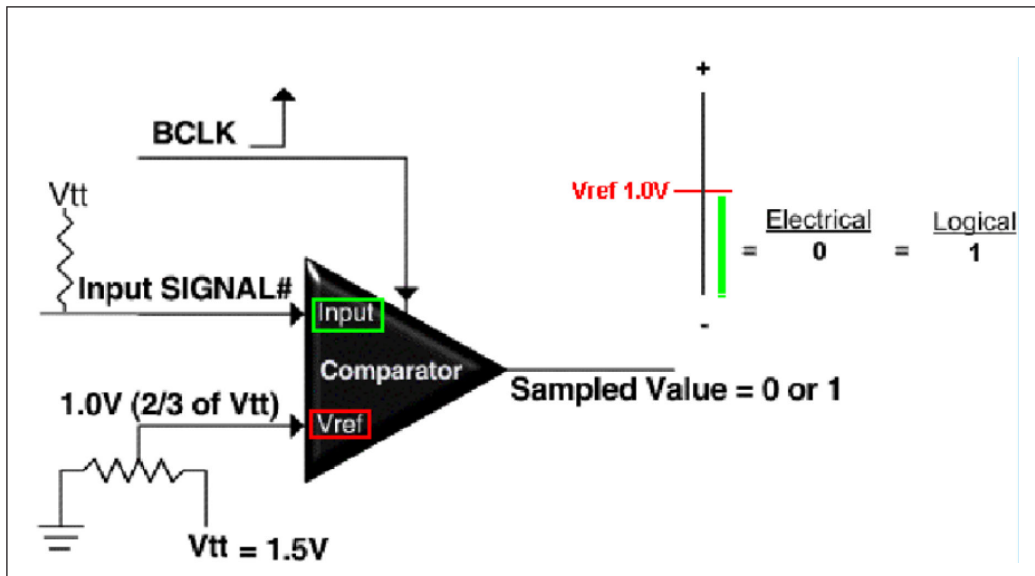
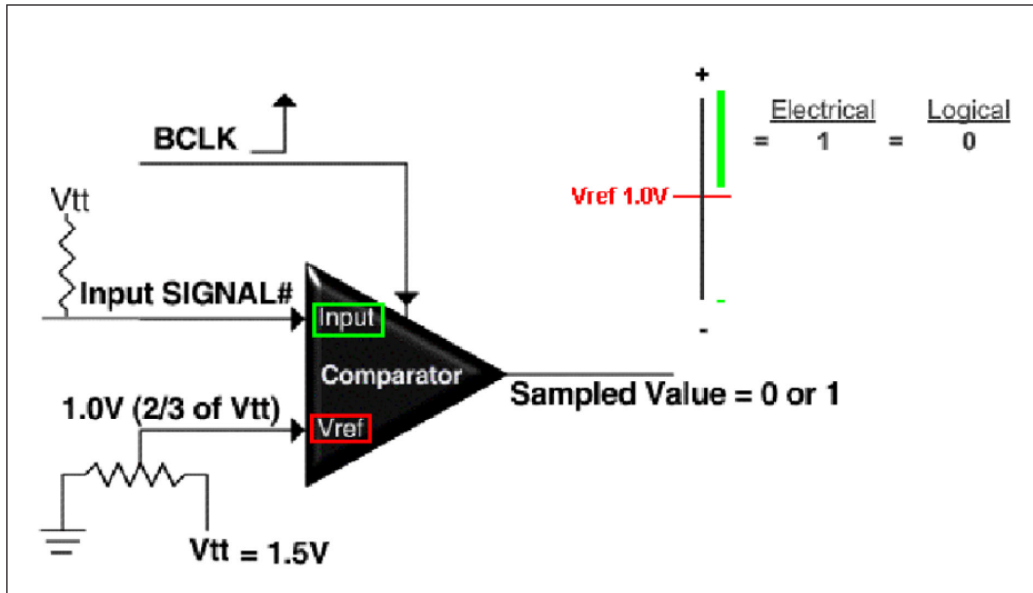


Figure 8-4: An Electrical Low = a Logical High



Chapter 8: P6 FSB Electrical Overview

Figure 8-5: An Electrical High = a Logical Low



All Signals Are Asserted When Low

All signals on the FSB are asserted when low. This includes the data bus and the address bus! The signals that comprise the address bus are designated as A[35:3]# (the # sign indicating that the signals are asserted when low). A logical one on an address signal is represented by an electrical high, and a logical zero as an electrical high. The same is true for the data bus, D[63:0]#. *Just as a reminder, Intel® frequently represents bus signal values in data book tables as logical values, not electrical values—so pay close attention to the fact that this is the reverse of their values when viewed on the bus.*

To place an electrical high on a GTL+ signal, don't drive it. The pull-up resistors keep the line high. In other words, the only time that a device actually drives a GTL+ signal is when it asserts it, thereby placing an electrical low on the line. To deassert a signal, turn off the output driver. The pull-ups pull the line back high.

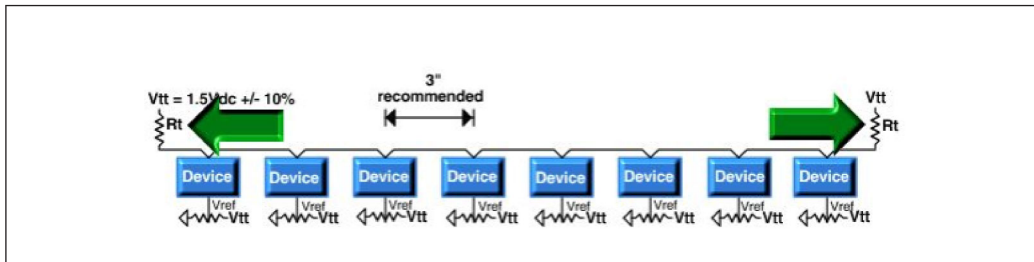
Pull-ups Return Lines Back to an Electrical High

The values chosen for the termination pull-ups used on either end of a GTL+ trace (see Figure 8-6 on page 210) are such that they return the line back to the electrically-high state fairly rapidly. The signals therefore have very fast edge rates and tend to overshoot and ring for some time before settling. The resistor values for the pull-ups on the ends of a trace match either:

- the effective impedance of the trace or
- the average impedance of all of the GTL+ trace runs.

The R_t (pull-up resistor value) is typically in the range from 25 to 65 ohms.

Figure 8-6: There Are Pull-ups at Both Ends of Each Trace



The Layout

Refer to Figure 8-7 on page 211. Each signal line is daisy-chained between bus agents and is pulled up to V_{tt} on both ends. No stubs are permitted when connecting each device to the trace run. In reality, however, it is impossible to have a completely stubless connection to each of the bus agents. The only stub permitted is the connection to each device's pin pad and each of these should not exceed 250ps (the maximum allowable flight time for the signal to traverse the stub) in length. Maintaining 3" $\pm$ 30% inter-device spacing (see Figure 8-8 on page 211) minimizes the variation in noise margins between networks (note that this is a recommendation, not a rule).

Chapter 8: P6 FSB Electrical Overview

Figure 8-7: Maximum Stub Length

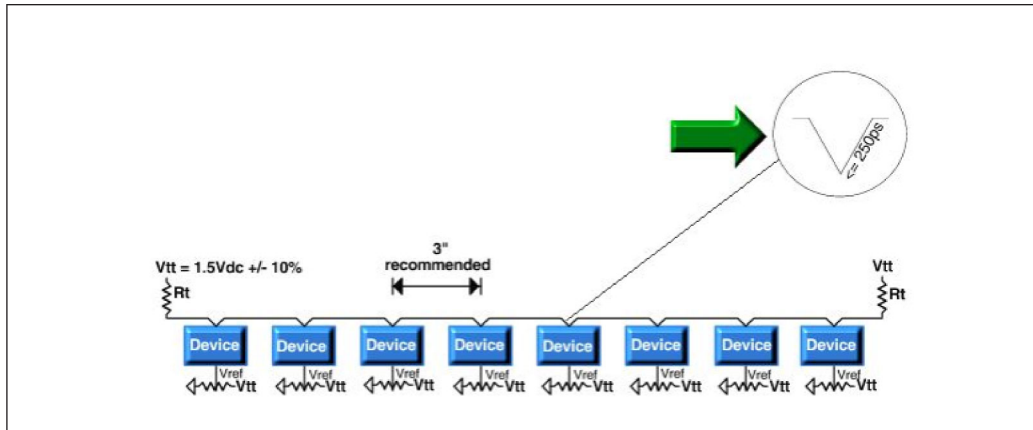
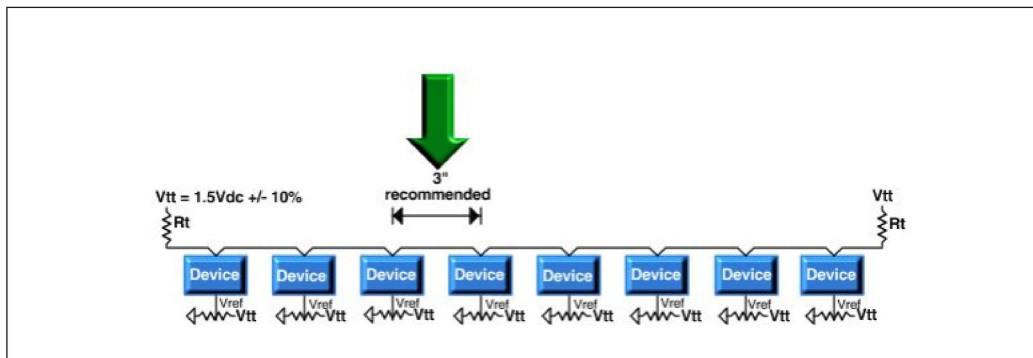


Figure 8-8: Inter-Device Spacing



Synchronous Bus

A device starts to drive a signal or a group of FSB signals on the rising-edge of BCLK and signals are only sampled on the rising-edge of the BCLK at the point where the BCLK rising-edge cross the V_{tt} level. BCLK is not a GTL+ signal. Rather, it is a 3.3V-tolerant signal and is supplied to all FSB agents from a central source. This ensures that all of the FSB devices are synchronized to the same clock.

How High is High and How Low is Low?

When the value at an input is sampled (when the rising-edge of the BCLK crosses the V_{tt} level), it is determined to be a **low** or **high** as follows:

- $-.3V_{dc} \leq \text{LOW} \leq V_{ref} - .2V_{dc}$ (see Figure 8-10 on page 213).
- $V_{ref} + .2V_{dc} \leq \text{HIGH} \leq V_{ccP}$ (see Figure 8-9 on page 212).

where $V_{ref} = 2/3$ of V_{tt} and V_{ccP} is the processor die's operating voltage. For the Pentium® Pro processor, the V_{ccP} was specified as follows:

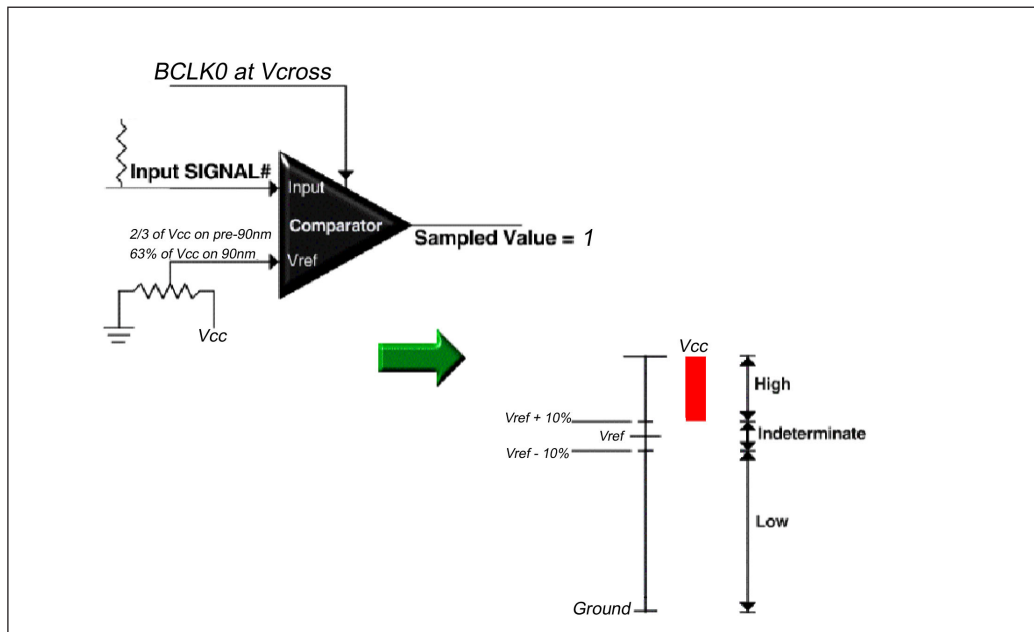
- For the 150Mhz processor with 256KB L2 Cache: $2.945V_{dc} \leq V_{ccP} \leq 3.255$, typical = $3.1V_{dc}$.
- For all other processor versions: $3.135V_{dc} \leq V_{ccP} \leq 3.465V_{dc}$, typical = $3.3V_{dc}$.

$V_{tt} = 1.5V_{dc} \pm 10\%$.

$V_{ref} = 2/3$ of $V_{tt} \pm 2\%$.

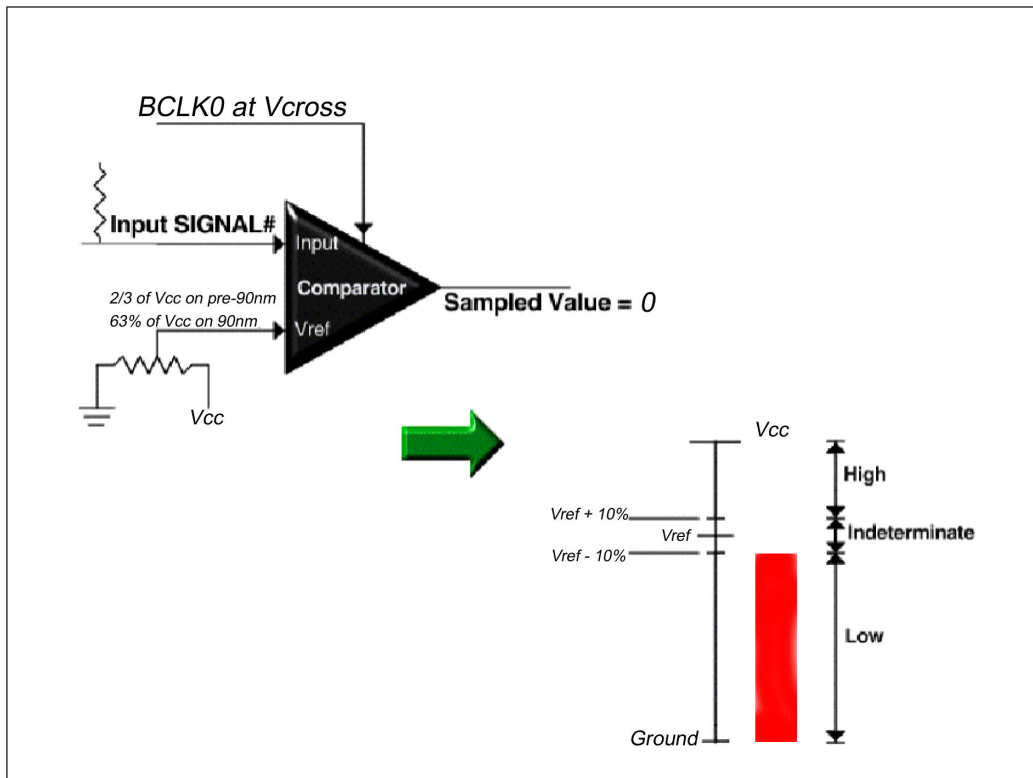
V_{ref} therefore = $1.0V_{dc}$.

Figure 8-9: Electrical High Limits



Chapter 8: P6 FSB Electrical Overview

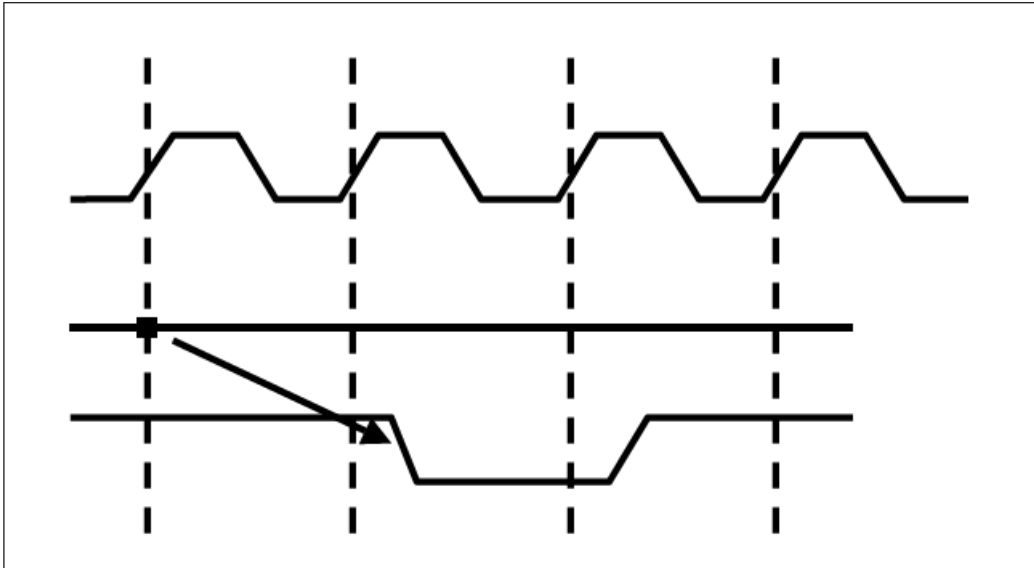
Figure 8-10: Electrical Low Limits



After Sampling, Minimum Response Time Is One Clock

The protocol is defined in a manner that always allows a minimum of one clock after a condition is detected before an agent must present a reply. As an example, refer to Figure 8-11 on page 214. Assume that a processor detects that it has acquired bus ownership on the rising-edge of a BCLK cycle. It takes action on this by assuming bus ownership starting on the rising-edge of the next BCLK cycle and driving out its transaction request onto the appropriate signals. It doesn't have to detect the condition on the rising-edge of a BCLK cycle and present its reply in the same BCLK so that it will be seen by others on the rising-edge of the next BCLK cycle.

Figure 8-11: Example



9

P6 FSB Arbitration Basics

Applicability to the Pentium® 4/Pentium® M

The Pentium® 4 and Pentium® M processors use the same FSB arbitration scheme as the P6 processor family. All of the topics introduced in this chapter therefore also apply to the Pentium® 4 and the Pentium® M processors.

Detailed Description of FSB Arbitration

For a detailed description of P6 FSB arbitration, refer to chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

What Signal Group are You Arbitrating For?

When an agent wins the arbitration, it takes ownership of the Request Phase signal group and uses it to issue a transaction request to the other agents that reside on the processor bus. A detailed account of the Request Phase of the transaction can be found in chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Symmetric Agent Arbitration—Democracy at Work

Introduction

A symmetric system is one in which any processor is capable of handling (i.e., executing) any task. The job of the SMP (symmetrical multiprocessing) OS is to attempt to keep all of the processors equally busy at all times (in other words, executing various tasks). At a given instant in time, one or more of the processors may require ownership of the Request Phase signal group in order to communicate with an external device. In a system designed for general purpose use, the bus arbitration scheme used to decide which of the processors gets ownership next is based on rotational priority—each of the processors has equal importance.

No External Arbiter Required

Figure 9-1 on page 217 and Figure 9-2 on page 218 identify the processors as the symmetric agents. The P6 processors that make up a cluster (i.e., the group of processors that reside on a FSB) have a built-in rotational priority scheme. No external arbitration logic is necessary to decide which of the processors that require the Request Phase signal group will attain ownership next. The arbiter incorporated within the FSB interface unit of each processor keeps track of:

- Whether any of the processors currently owns the Request Phase signal group.
- Which of them owned the signal group last (or still owns it).
- Which of them gets to use it next (assuming it is asking for ownership).

In order for the processors to track this information, each must know its own agent ID as well as the agent ID of the processor that last gained ownership of the Request Phase signal group. If it is known who had ownership last (or still has it), then it is also known the agent ID whose turn it is next (because it's a rotational scheme).

Chapter 9: P6 FSB Arbitration Basics

Figure 9-1: System Block Diagram (Single Processor Cluster)

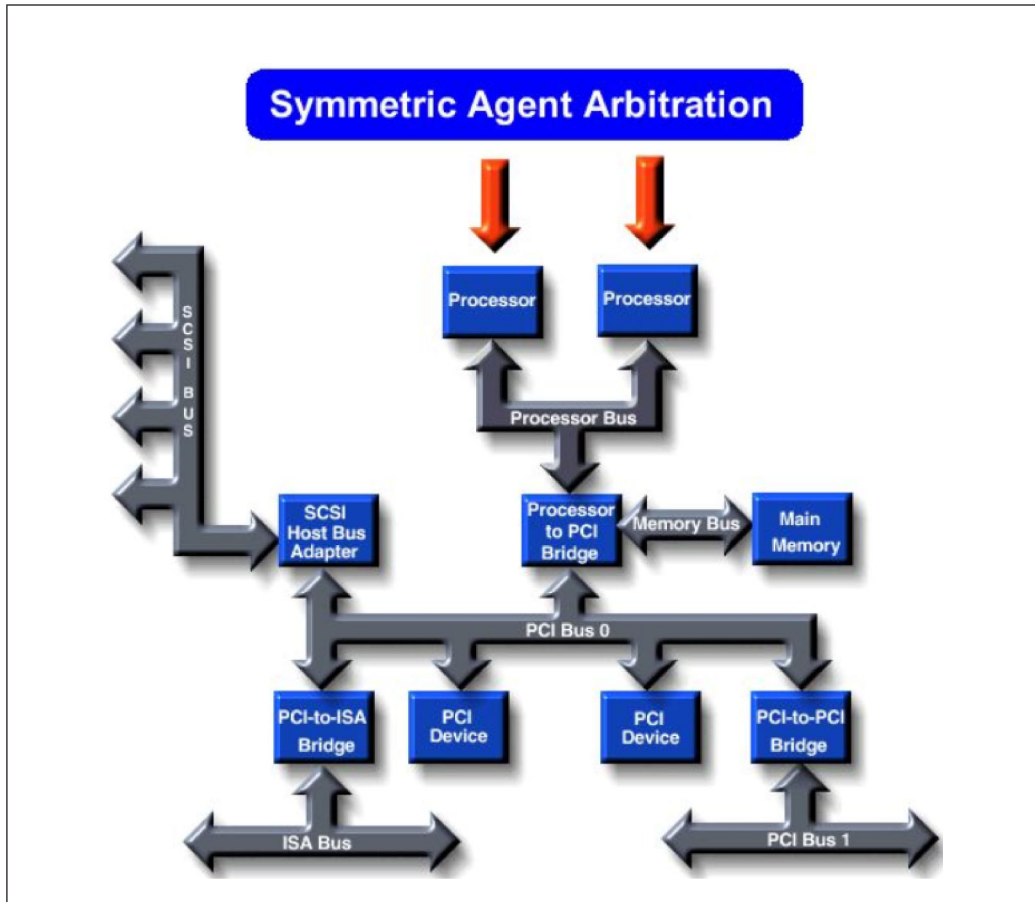
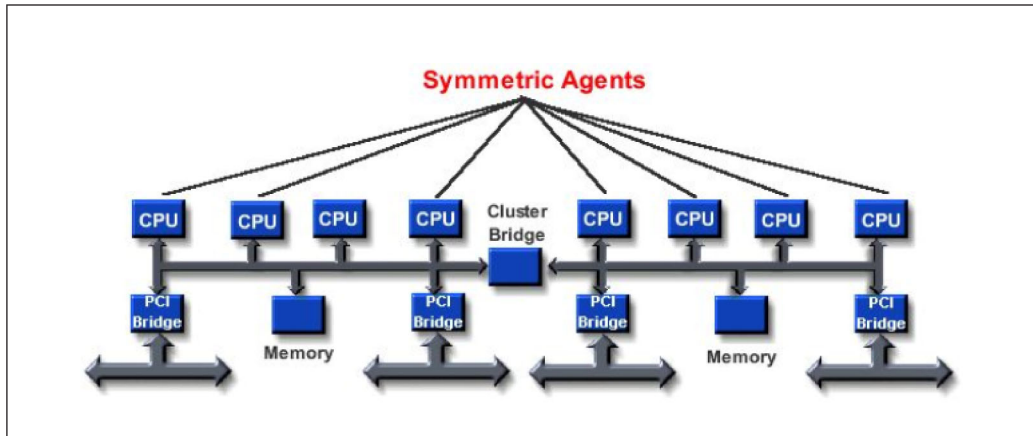


Figure 9-2: System Block Diagram (Multiple Processor Clusters)



Agent ID Assignment

Each processor is assigned its Agent ID automatically at the trailing-edge of reset when it samples the state of its Bus Request inputs:

- A Pentium® Pro or a Xeon MP samples its BR[3:1]# inputs to determine its agent ID.
- A Pentium® II or a Pentium® III samples its BR[1]# input to determine its agent ID.
- A Celeron processor does not sample its inputs. Rather, it is automatically assigned an Agent ID of 0.

A detailed description can be found in "Assignment of IDs to the Processor" on page 860 in the book.

Arbitration Algorithm

Rotating ID

As stated earlier, each processor must track which of them gained Request Phase signal group ownership last. This is referred to as the *Rotating ID*. When reset is asserted, the Rotating ID is reset to one of the following values:

Chapter 9: P6 FSB Arbitration Basics

- 3 in a Pentium® Pro or Xeon MP processor (see Figure 9-3 on page 219).
- 1 in a Pentium® II or Pentium® III processor.
- 1 in a Celeron processor.

The processors think that the processor whose ID matches the Rotating ID owned the Request Phase signal group last and therefore processor 0 should get ownership next (if it asks for it). In a Pentium® Pro or Xeon MP system, the sequence (see Figure 9-4 on page 220) in which the processors gain ownership (if all of the processors were asking for ownership after reset was deasserted) would be 0, 1, 2, 3, 0, 1, etc.

Figure 9-3: Reset Initializes the Rotating ID

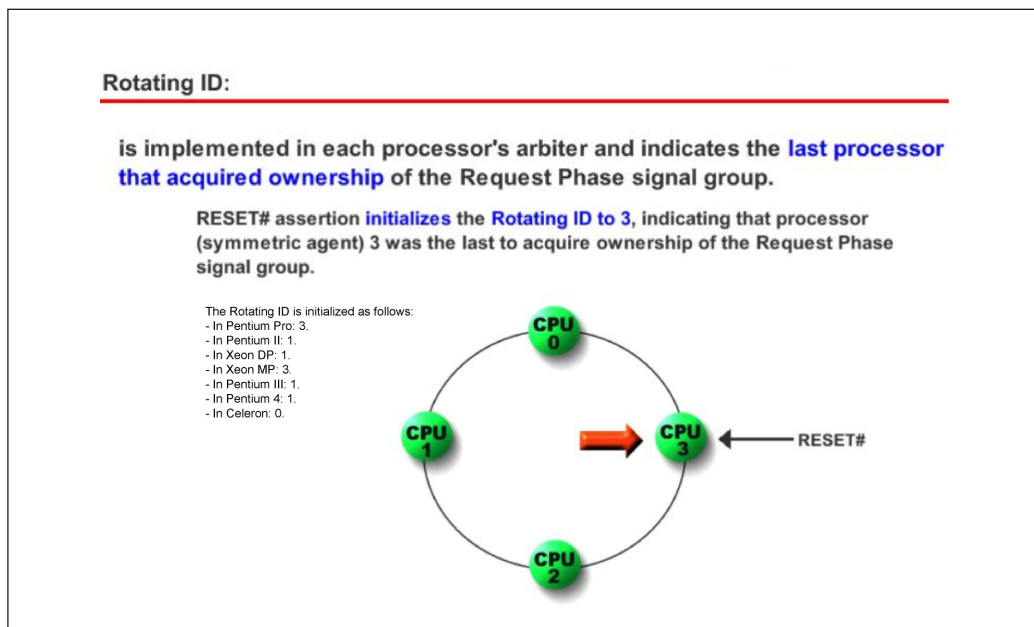
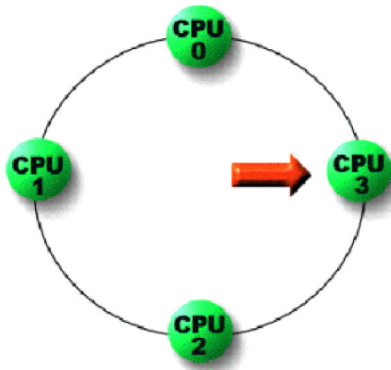


Figure 9-4: Ownership Passes from CPU to CPU in a Strict Rotation

Rotating ID:

When more than one processor is requesting ownership of the Request Phase signal group, ownership passes from processor to processor in a **strict rotation**.



Busy/Idle State

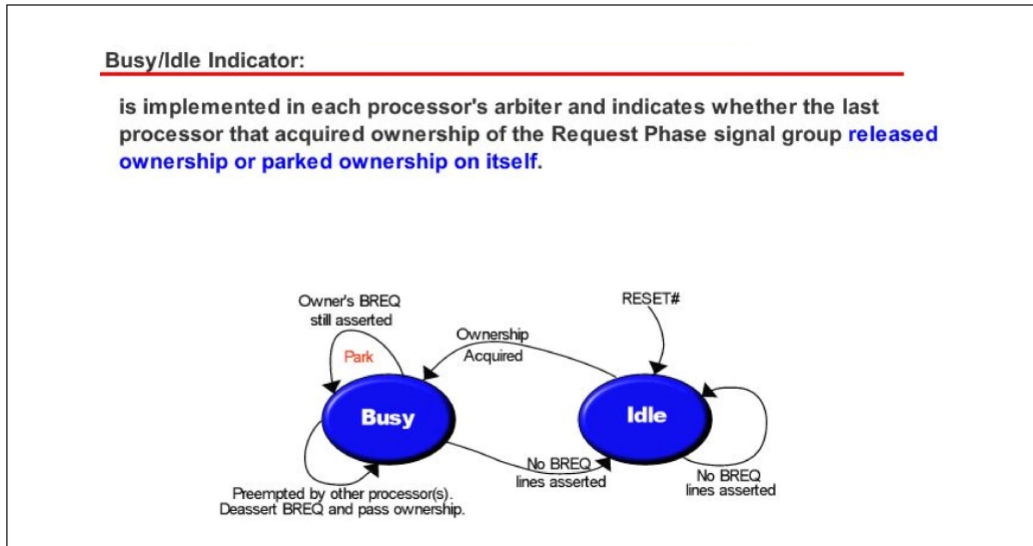
In addition to the Rotating ID, each processor must also keep track of whether the last processor that gained ownership of the Request Phase signal group retained ownership or has released it (and therefore none of them currently owns it). When the last owner retains ownership, the ownership state is said to be *Busy*. If the previous owner surrendered ownership and none of them currently owns the Request Phase signal group, the ownership state is said to be *Idle*. Each of the processors maintains an internal state indicator (see Figure 9-5 on page 221) to indicate whether the Request Phase signal group ownership state is currently Busy or Idle.

It is incorrect to think of Busy as meaning that one of the processors currently owns the Request Phase signal group and is using it. Rather, it only means that the processor has retained ownership—it may or may not currently be using the Request Phase signal group.

A detailed description of the Busy/Idle indicator may be found in “The Busy/Idle Indicator” on page 1153 in the book.

Chapter 9: P6 FSB Arbitration Basics

Figure 9-5: The Busy/Idle Indicator



Bus Parking

The processor may retain ownership after completing a transaction (in case it may need the Request Phase signal group again in the future—note that the P6 processors do this). This is referred to as parking ownership on yourself. If a processor is successful at retaining ownership until such time as it may need to issue another transaction request, parking provides faster access to the Request Phase signal group (a request needn't be made for the current symmetric owner to surrender ownership).

Be Fair!

When a processor parks ownership on itself, it may retain ownership until another processor requests ownership (i.e., until it detects the assertion of another processor's Bus Request—BREQ—signal). In other words, be fair to the other processors—don't hog the bus.

Requesting Ownership

Figure 9-6 on page 222 illustrates a quad-processor Pentium® Pro or Xeon system (i.e., with four processors residing on the same FSB). The following discussion assumes this FSB processor population model. The Priority agent in the

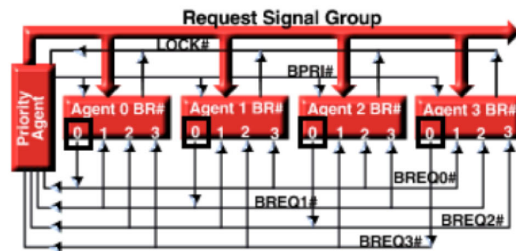
The Unabridged Pentium® 4

illustration represents a device other than a processor that may need to use the FSB to perform transactions (e.g., the North Bridge—see Figure 7-1 on page 195).

There is a temptation to think that agent 0 (i.e., the processor with an agent ID of 0) uses its BR0# output to request ownership, that agent 1 uses its BR1# output, etc., but it's not true. When any processor wants to issue a new transaction request, it uses its bus request output pin, BR0#, to request ownership. BR[3:1]# are processor inputs that are sampled on the rising-edge of each BCLK cycle to see if any of the other processors are requesting ownership. If agent 0 is requesting ownership of the FSB, the BREQ0# signal line is asserted. If agent 1 is requesting ownership, the BREQ1# signal line is asserted, etc. The interconnect of the BREQ signal lines on the system board that is shown in Figure 9-6 on page 222 and Figure 9-7 on page 223 is mandated by the protocol specification. Although not shown, the system board designer includes a pull-up resistor on each of the four traces.

Each of the processors knows which of the BREQn# signal lines belongs to it and which belongs to each of the other processors.

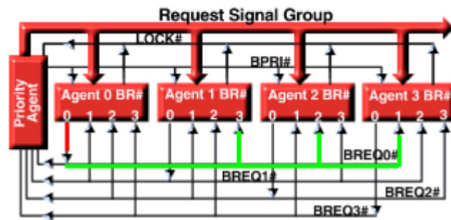
Figure 9-6: Quad-Processor Pentium® Pro System



Each processor has a single Bus Request output pin (BR0#).

Chapter 9: P6 FSB Arbitration Basics

Figure 9-7: Bus Request Signal Interconnect



It is connected to the BR[3:1]# input pins on the other 3 processors.

Detailed Description of P6 Symmetric Arbitration

A detailed example of P6 processor FSB arbitration can be found in chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Priority Agent Arbitration

Request to Break Into the Rotation

When a device other than a processor must initiate a transaction on the FSB, it asserts the BPRI# (Bus Priority Agent Request) signal to the processors that reside on the FSB. The assertion of BPRI# requests that the processors permit the Priority agent to break into the rotational priority scheme so it can be the next entity to issue a transaction request. Most of the time, the processors will grant this request.

There is one case where the Priority agent's request will not be honored until the processor that currently owns the FSB completes a locked series of transactions. An introduction to this topic can be found in "Locking—Shared Resource Acquisition" on page 226. A detailed description of this topic can be found in "Pentium® 4 Locked Transaction Series" on page 1177 in the book.

Example Priority Agents

While the symmetric agents are very polite to each other, there may be one or more devices on the FSB capable of performing FSB transactions but that are not processors. Such agents are referred to as Priority Agents. Refer to Figure 9-8 on page 225. This figure illustrates a system with two FSBs connected to each other by an inter-cluster bridge. Each of the FSBs is populated by:

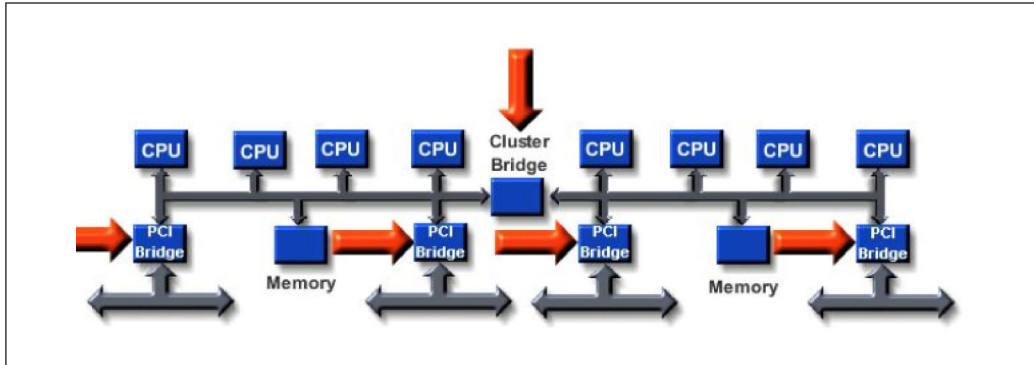
- The inter-cluster bridge. When a processor or a FSB-to-PCI Bridge on either FSB attempts to access memory that resides on the other FSB, the inter-cluster bridge must act as its surrogate on the target FSB. To do so, it asserts BPRI# (Bus Priority agent Request) to the array of processors on the target FSB.
- Four Pentium® Pro or Xeon processors. These are Symmetric rather than Priority agents.
- Two FSB-to-PCI Bridges. When a PCI Bus Master that resides beneath the FSB-to-PCI Bridge requires access to system memory on either FSB, the FSB-to-PCI Bridge must act as the PCI Bus Master's surrogate on the FSB. To do so, it asserts BPRI# (Bus Priority agent Request) to the array of processors on the FSB immediately connected to the FSB-to-PCI Bridge.
- A system memory module. It never initiates transactions on the FSB. Rather, it acts as the target of FSB transactions initiated by Request Agents on the FSB.

BPRI# Is Not a Shared Signal

The protocol permits only one device to assert BPRI# at a time. If there are multiple Priority Agents, there must be a chipset-specific method for the Priority Agents to arbitrate amongst themselves to determine which of them gets to use BPRI# to request ownership (if more than one of them needs to issue a transaction request at the same time).

Chapter 9: P6 FSB Arbitration Basics

Figure 9-8: Dual Cluster, Eight-Way System Block Diagram



Priority Agent Parking

The FSB specification states that the Priority Agent can keep BPRI# asserted when it initiates its final (or only) transaction request. The exact wording is “provided it can guarantee forward progress of the symmetric agents.” In other words, be fair to the common people. If the Priority Agent has parked ownership on itself by keeping it BPRI# asserted, it can retain ownership until:

- it has to initiate another transaction request, or
- it samples any of the BREQn# lines asserted, indicating that one or more of the symmetric agents (i.e., the processors) require access to the Request Phase signal group,

whichever comes first.

Be Fair to the Common People

The FSB specification dictates that the Priority Agent must keep BPRI# deasserted for at least two clocks after its last deassertion of BPRI#. This opens a window that permits the symmetric agents to get ownership of the Request Phase signal group if any of them need to issue a transaction request.

Detailed Description of P6 Priority Agent Arbitration

A detailed description of P6 Priority Agent arbitration can be found in chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*.

Locking—Shared Resource Acquisition

Introduction

The previous section, “Priority Agent Arbitration” on page 223,” mentioned that there is one case wherein a Priority Agent’s request for ownership of the FSB is not honored immediately by the processors on the FSB.

This section describes the circumstances under which a processor would need to perform a series of transactions with a guarantee that no other agent can use the FSB until its transaction series has completed.

Shared Resource Concept

Assume that the OS sets aside an area of memory to be used by tasks executing on multiple processors (or even by different tasks executed by the same processor) as a shared memory buffer. It is intended to be used as follows:

1. Before using the memory buffer (i.e., reading from or writing to it), a task must first test a flag to ensure that the buffer isn’t currently owned by another task. If the flag indicates that the buffer is currently in use, the task wishing to gain ownership should periodically check the flag to determine when it becomes available.
2. When the flag indicates that the buffer is available (i.e., the flag location contains 0), the task sets the flag to a non-zero value indicating that it has exclusive ownership of the buffer. The buffer is then unavailable if any other task should test the flag.
3. Having gained exclusive ownership of the buffer, the task can now read and write the buffer.
4. If the buffer is in an area of memory designated as WT (cacheable Write Through), WC (non-cacheable Write Combining), WP (Write Protect), or UC (Uncacheable) memory (refer to “The Memory Types” on page 581 in the book), memory writes are absorbed into the processor’s internal Posted Write Buffers. Although a processor performs a lookup (i.e., a snoop) in its internal caches when other agents access memory, *the Posted Write Buffers are not snooped*. In this case, when the task is done using the buffer, it should ensure that all of its updates (i.e., memory writes) have been flushed to memory from the Posted Write Buffers.
5. After ensuring that the buffer has received all updates, the task should release ownership of the buffer (i.e., clear the flag) so the buffer can be used by other tasks.

Testing Availability and Gaining Ownership of a Shared Resource

The OS typically uses a memory location (or series of memory locations) as the flag indicating the availability or unavailability of a particular shared resource. This is referred to as a *memory semaphore*. It is used as follows:

1. Before using the buffer (i.e., reading from or writing to it), a task reads the semaphore associated with the buffer (via a memory read) to ensure that the buffer isn't currently being used by another task. If the buffer is currently unavailable (usually indicated by a non-zero semaphore value), the task wishing to gain ownership should periodically check the semaphore location to determine when it becomes available.
2. When the flag indicates that the buffer is available (the semaphore = 0), the task writes a non-zero value into the semaphore to indicate that a task now has exclusive ownership of the buffer. The buffer will then be marked as unavailable if any other task should test the semaphore.
3. Having gained exclusive ownership of the buffer, the task can now read and write the buffer.
4. If the memory buffer area is designated as WC, WT, WP, or UC memory, when the task is done using the buffer, it should ensure that any buffer updates (i.e., memory writes) are flushed from the processor's Posted Write Buffers to memory.
5. After ensuring that the buffer has received all updates, the task releases ownership of the buffer (by clearing the semaphore to zero) so the buffer can be used by other tasks.

Race Condition Can Present Problem

Consider the following possibility:

1. The task executing on processor 0 performs a memory read transaction on the FSB to read the semaphore and determine the buffer's availability.
2. The task tests the semaphore's value and determines that the buffer is available (semaphore = 0).
3. After the task on processor 0 has completed the memory read to obtain and test the semaphore value, a task executing on processor 1 has initiated a memory read request on the FSB to test the state of the same semaphore. It completes the read and begins testing the value.

4. The processor 0 task initiates a memory write transaction on the FSB to update the semaphore to a non-zero value, thereby marking the shared buffer as unavailable. After it completes the write on the FSB, it considers itself the sole owner of the buffer.
5. The processor 1 task also determined that the buffer is available. It now uses the FSB to perform a memory write transaction updating the semaphore to a non-zero value. This marks the shared buffer as unavailable. It completes the write and considers itself the sole owner of the buffer.

Two tasks executing on two separate processors now each believe that they have exclusive ownership of the buffer and problems will ensue. If either of the tasks reads from the buffer (by performing a memory read on the FSB), its memory read is not snooped in the other processor's Posted Write Buffers. If any updates have been logged in the other processor's Posted Write Buffers, this will not be signaled to the processor performing memory read or to the system memory controller. The requested read data is therefore sourced by system memory and may contain stale data.

Guaranteeing Atomicity of Read/Modify/Write

The problem outlined in the previous section came about because processor 1 was able to read the semaphore immediately after processor 0 read it. The two processors were in a race condition checking the semaphore. Processor 0 then wrote to it, followed immediately by processor 1 writing to it. The tasks on the two processors each ended up believing it had sole ownership of the buffer.

The problem can be prevented if processor 0 could prevent other Request Agents from using the FSB from the time it initiates its semaphore read until the time it updates the semaphore to a non-zero value. In other words, it should lock the FSB while it performs the semaphore read/modify/write (frequently referred to as a RMW).

To do this, the programmer uses a special instruction to perform the RMW operation. When using such instructions:

1. The processor asserts the LOCK# signal when it initiates the semaphore memory read, keeps LOCK# asserted while it performs the internal test of the semaphore value, and performs the memory write to update the semaphore before releasing the LOCK# signal. The continued assertion of LOCK# prevents any Priority Agent from obtaining FSB ownership during this period.
2. The processor also keeps its BREQ0# output pin asserted throughout this period to keep any of the other processors from obtaining ownership of the Request Phase signal group to start a transaction.

Detailed Description of P6 FSB Locking

A detailed description of the P6 processor's locking mechanism can be found in chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Blocking New Requests—Stop! I'm Full!

Introduction

Refer to Figure 9-9 on page 230. The section entitled “Transaction Tracking” on page 203 introduced the concept of transaction tracking and the IOQ (In Order Queue). Each agent has an IOQ (see Figure 9-10 on page 231) that it uses to keep track of each transaction that is currently outstanding on the FSB (irrespective of which agent initiated the transactions). The depth of an agent's IOQ is device-specific. The P6 processor has a selectable queue depth of either one or eight. The queue depths of the chipset agents is also design-specific. Low-end Intel® P6 chipsets typically had a queue depth of four, while the high-end chipsets had a queue depth of eight.

Obviously, each agent has a finite number of entries in its IOQ and this defines the number of transactions the agent can track on the FSB during a given period of time. Each agent must have the ability to throttle the ability of other agents to initiate new transactions. That is the purpose of the BNR# (Block Next Request) signal. An agent must assert BNR# when its In Order Queue (IOQ) has one entry remaining empty. This is necessary because a new transaction request could be issued by another agent at the same time that an agent begins to assert BNR#. The one entry that is remaining can then be used to latch and track the newly-issued transaction. There is no danger that another transaction will be issued to the FSB because all agents have detected BNR# by this time.

Figure 9-11 on page 231 illustrates what would happen if an agent (e.g., a P6 processor) waited until its IOQ was full before it asserted BNR#. If another transaction were issued by any agent while the agent with the queue-full condition asserts BNR#, that agent could not latch and track that transaction. From that point forward, that agent is incapable of interacting with the FSB (because it wouldn't know when it was safe to use the signal groups associated with each transaction phase).

The Unabridged Pentium® 4

Refer to Figure 9-12 on page 232. All agents that are capable of initiating transactions on the FSB must monitor the BNR# signal.

BNR# could also be used by a debug tool to create a controlled situation where no additional transactions can be issued to the FSB until the current transaction has been completed. In other words, transactions could be single-stepped onto the bus to simplify the debug process.

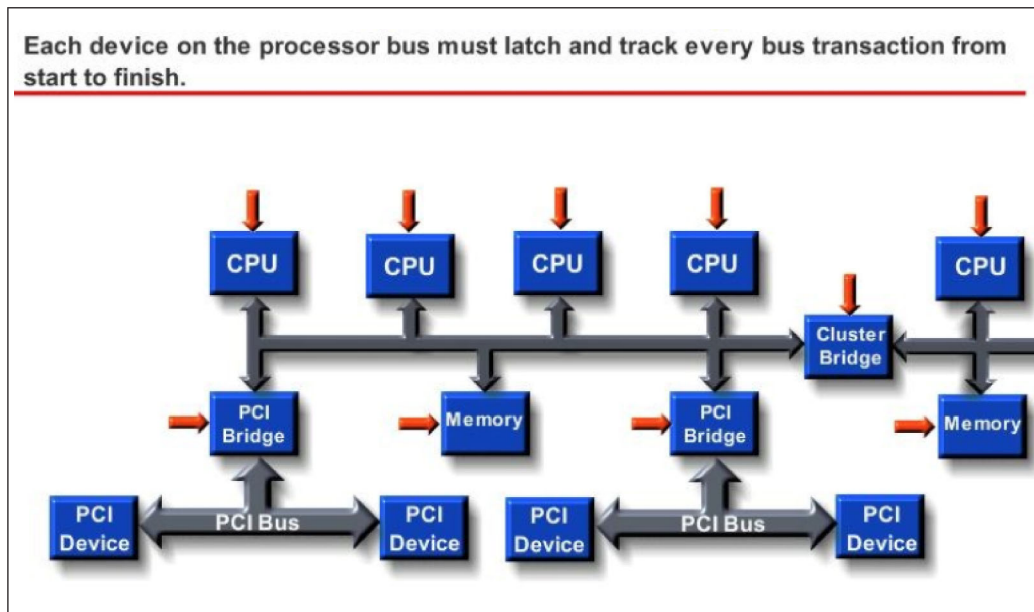
BNR# is Shared Signal

BNR# is a shared, open-drain signal because multiple bus agents may assert it simultaneously to indicate that they are not ready to deal with a new transaction.

Detailed Description of P6 Transaction Blocking

A detailed description of P6 transaction blocking can be found in chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

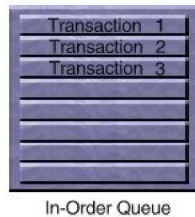
Figure 9-9: All FSB Agents Must Track FSB Activity



Chapter 9: P6 FSB Arbitration Basics

Figure 9-10: The In Order Queue

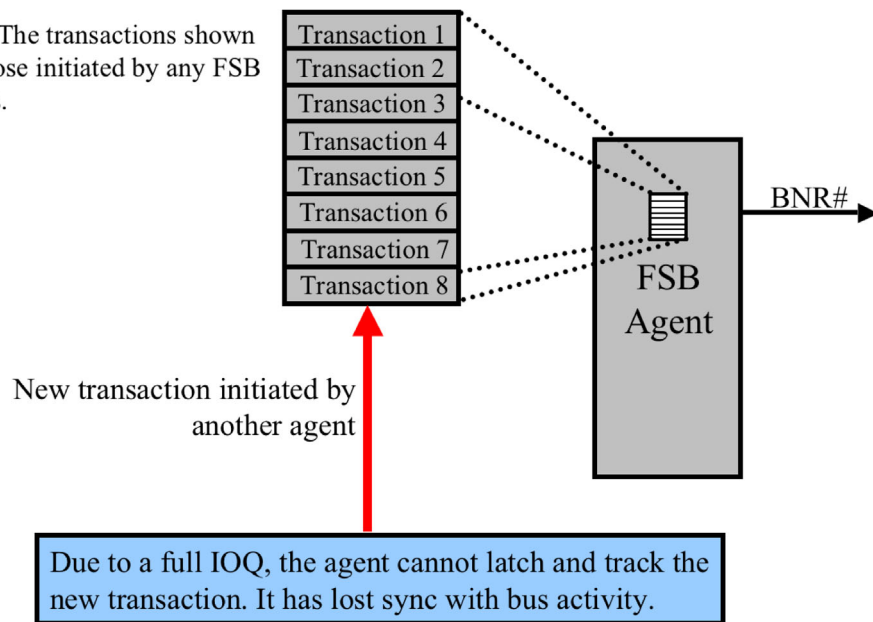
This requires the implementation of an In-Order Queue (IOQ) in each device.



The depth of a device's IOQ is design-specific

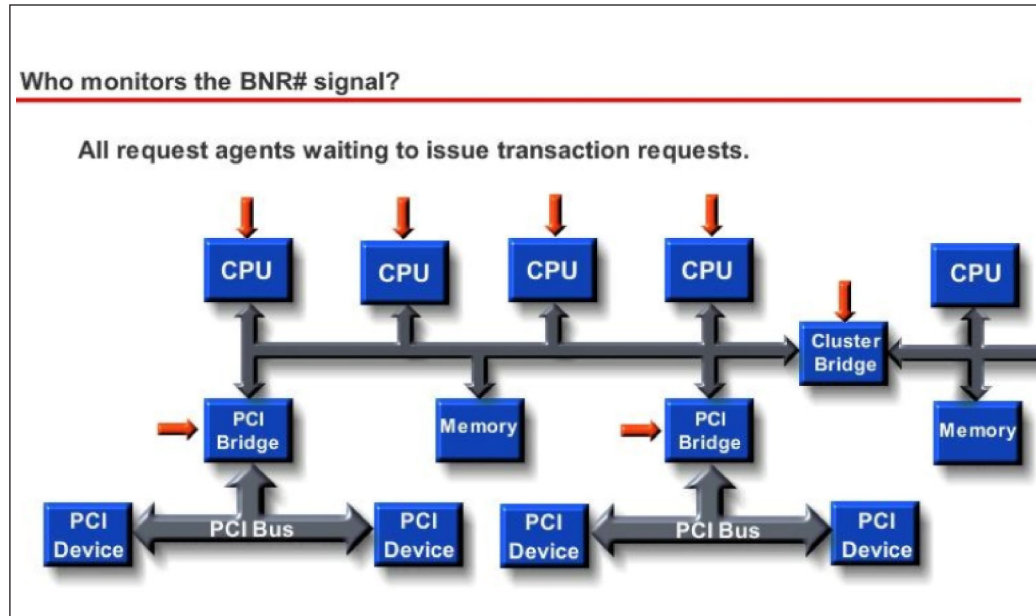
Figure 9-11: The In Order Queue and BNR#

Note: The transactions shown are those initiated by any FSB agents.



The Unabridged Pentium® 4

Figure 9-12: All Agents Capable of Initiating Transactions Must Monitor BNR#



10

P6 FSB Request & Error Phase Basics

Applicability to the Pentium® 4/Pentium® M

The Pentium® 4 Request Phase Is Double-Pumped

In the Request Phase of a transaction, the Request Agent outputs all of the information that describes the transaction request. This information is output in two packets:

- In the P6 FSB protocol, the Request Agent outputs the two packets of request information in the first two BCLK cycles of the transaction. All FSB agents latch the first packet on the rising-edge of the second BCLK cycle, and the second packet on the rising-edge of the third BCLK cycle of the transaction.
- Starting with the Pentium® 4 processor, the two packets are both output during the first BCLK cycle of the transaction. The Request Agent sourcing the two packets also sources two strobe signals that the other FSB agents use to latch the two packets into their input receivers.

The Pentium® 4 FSB protocol therefore delivers the two packets that comprise the request in a single BCLK cycle.

Request Phase Parity Scheme Was Altered

The parity protection scheme used in the Pentium® 4 FSB protocol was changed. A detailed description of the P6 Request Phase parity protection scheme can be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition. A detailed description of the Pentium® 4 Request Phase parity protection scheme can be found in “Request Phase Parity Checking” on page 1205 in the book.

The Error Phase Was Eliminated in the Pentium® 4

With the advent of the Pentium® 4 processor, the Error Phase was eliminated from the transaction. The Pentium® M, the Pentium® 4 Celeron, and the Pentium® 4 Xeon processors use the same FSB protocol as the Pentium® 4.

Introduction to the Request Phase

A Request Consists of a Large Amount of Information

On earlier IA32 processor FSB implementations, a transaction request consisted of a very limited amount of information:

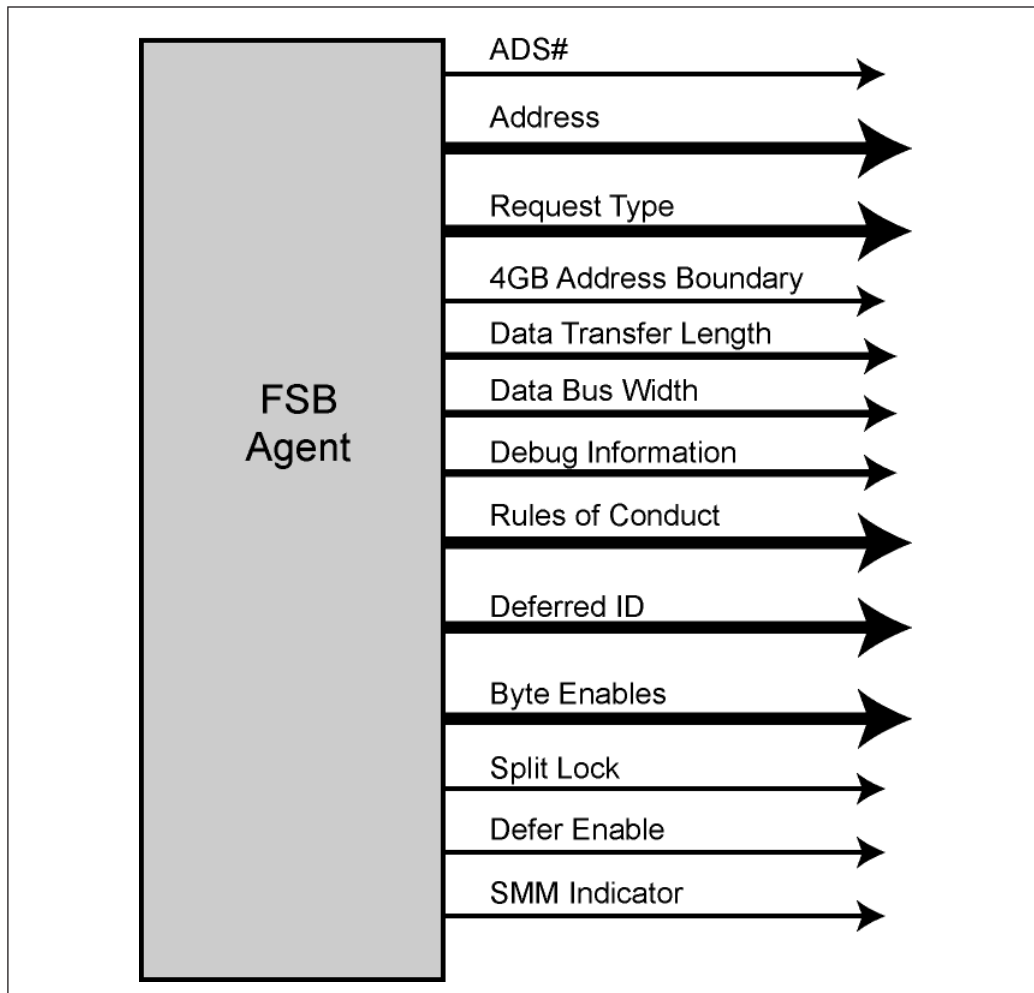
- The memory or IO address that selected the target memory or IO dword or qword.
- The Byte Enables that indicated which bytes in the currently addressed memory or IO dword or qword were to be transferred.
- The transaction type (e.g., a memory read or an IO write).
- The type of memory being addressed (e.g., cacheable WT, cacheable WB, or uncacheable memory). Note that this item of information was not output on the 386 or early 486 FSB. It was output on the Pentium® and the version of the 486 that implemented a WB L1 Data Cache.

Refer to Figure 10-1 on page 236. In the P6 FSB protocol, however, a transaction request consists of a relatively large amount of information:

Chapter 10: P6 FSB Request & Error Phase Basics

- The ADS# signal is asserted to indicate that a new transaction has been initiated.
- The memory or IO address that selects the target memory or IO qword.
- The Byte Enables that indicate which bytes in the currently-addressed memory or IO qword are to be transferred.
- The transaction type (e.g., a memory read or an IO write).
- If it's a memory transaction, whether or not the address is below or above the 4GB address boundary.
- If it's an IO or a memory transaction, the amount of data that is to be transferred.
- The width of the Request Agent's data bus.
- Optional, Intel®-confidential debug information.
- If it's a memory transaction, the type of memory being addressed.
- The Request Agent type (i.e., Symmetric or Priority Agent) and its Agent ID.
- The LOCK# signal to indicate whether this transaction is part of a locked series of transactions.
- The Split Lock signal to indicate whether or not the semaphore being addressed straddles a qword address boundary.
- Whether or not the Request Agent grants the Response Agent permission to defer the transaction.
- If it's a memory transaction, whether regular or SMM memory is being addressed.

Figure 10-1: A Transaction Request Consists of a Large Amount of Information



Introduction to the Request Phase Signal Group

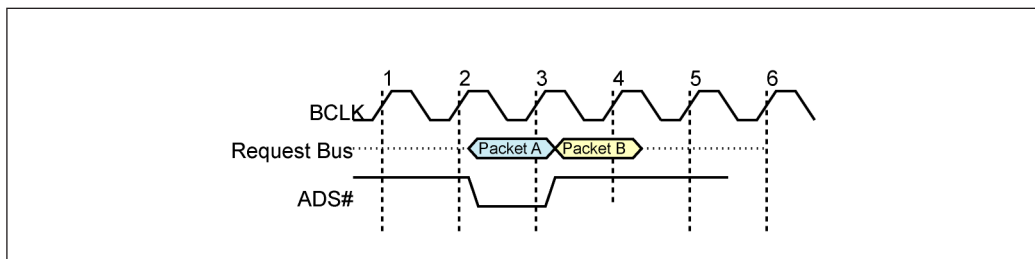
Once ownership of the Request Phase signal group has been acquired (see “P6 FSB Arbitration Basics” on page 215), the Request Agent uses the Request Phase signal group to broadcast the transaction request to all of the other FSB agents. This includes the address and transaction type, as well as additional information about the transaction. The **Request Phase signal group** consists of the following signals:

Chapter 10: P6 FSB Request & Error Phase Basics

- **A[35:3]#**. Address bus.
- **AP[1:0]#**. Address bus parity bits. As noted earlier, starting with the Pentium® 4, a different parity scheme is used.
- **REQ[4:0]#**. Request bus.
- **RP#**. Parity bit for REQ[4:0]# and ADS#. As noted earlier, starting with the Pentium® 4, a different parity scheme is used.
- **ADS#**. Address Strobe.

As mentioned earlier, the Request Phase is always two clocks in duration and the information about the transaction is output in two packets (see Figure 10-2 on page 237), during the first two BCLK cycles of the transaction. ADS# is asserted during the first clock and deasserted during the second. Its assertion indicates that a new transaction request is being broadcast. All FSB agents, not only Response Agents (i.e., potential targets), latch both packets. As discussed earlier, all agents must track the transaction as it passes through each phase from initiation to completion. In addition, if it is a memory transaction, Snoop Agents (i.e., processors with internal caches) must submit the memory address to their caches for a lookup and must deliver the snoop result during the transaction's Snoop Phase. The potential Response Agents must decode the address and transaction type to determine which of them is the target of the transaction.

Figure 10-2: Two Information Packets Broadcast during Request Phase



Request Phase Signal Group is Multiplexed

There is more transaction-related information to be output during the Request Phase than there are pins in the Request Phase signal group. The same signal group is used during each of the two clocks that comprise the Request Phase, but different information is output on these pins during each clock. Intel® refers to the two information packets as packets A and B.

The Unabridged Pentium® 4

In the Intel® data book, the information output in the two packets is referred to in the following manner:

- See Figure 10-3 on page 239. The information output on the address bus (A[35:3]#) and the request bus (REQ[4:0]#) signals during the transmission of Packet A is referred to as Aa[35:3]# and REQa[4:0]#. In the illustration, all FSB agents latch packet A on the rising-edge of BCLK 3.
- See Figure 10-4 on page 240. The information output on the address bus (A[35:3]#) and request bus (REQ[4:0]#) signals during the transmission of Packet B is referred to as Ab[35:3]# and REQb[4:0]#. In the illustration, all FSB agents latch packet B on the rising-edge of BCLK 4.

Packet A Content

The information output during the transmission of packet A consists of the memory or IO address on A[35:3]# and the transaction type on REQ[4:0]#. If it's a memory transaction, the Request Agent also indicates whether or not the address is above or below the 4GB address boundary.

Packet B Content

The information output during the transmission of Packet B are actually internal signals that are gated onto the address and request pins during the second clock of the transaction. Table 10-1 on page 238 defines the names of the internal signals driven onto A[35:3]# and REQ[4:0]# during the second clock.

Table 10-1: Packet B Signal Names

Output Pin(s)	Signal Names
A[35:32]#	Intel®-confidential optional debug information.
A[31:24]#	ATTR[7:0]# . Attribute signals. See “The Contents of Request Packet B” on page 1219 in the book.
A[23:16]#	DID[7:0]# . Deferred ID. See “The Contents of Request Packet B” on page 1219 in the book.
A[15:8]#	BE[7:0]# . The eight Byte Enable signals indicate which of the bytes in the currently-addressed qword are to be transferred (i.e., read or written).

Chapter 10: P6 FSB Request & Error Phase Basics

Table 10-1: Packet B Signal Names (Continued)

Output Pin(s)	Signal Names
A[7:3]#	EXF[4:0]# . Extended function signals. See “The Contents of Request Packet B” on page 1219 in the book.
REQ[4:0]#	In a Deferred Reply transaction, this field has no meaning. In other transaction types, these signals provide additional transaction definition including (but not limited to): • The width of the Request Agent’s data bus. • The amount of data to be transferred.

Figure 10-3: Packet A Signal Names

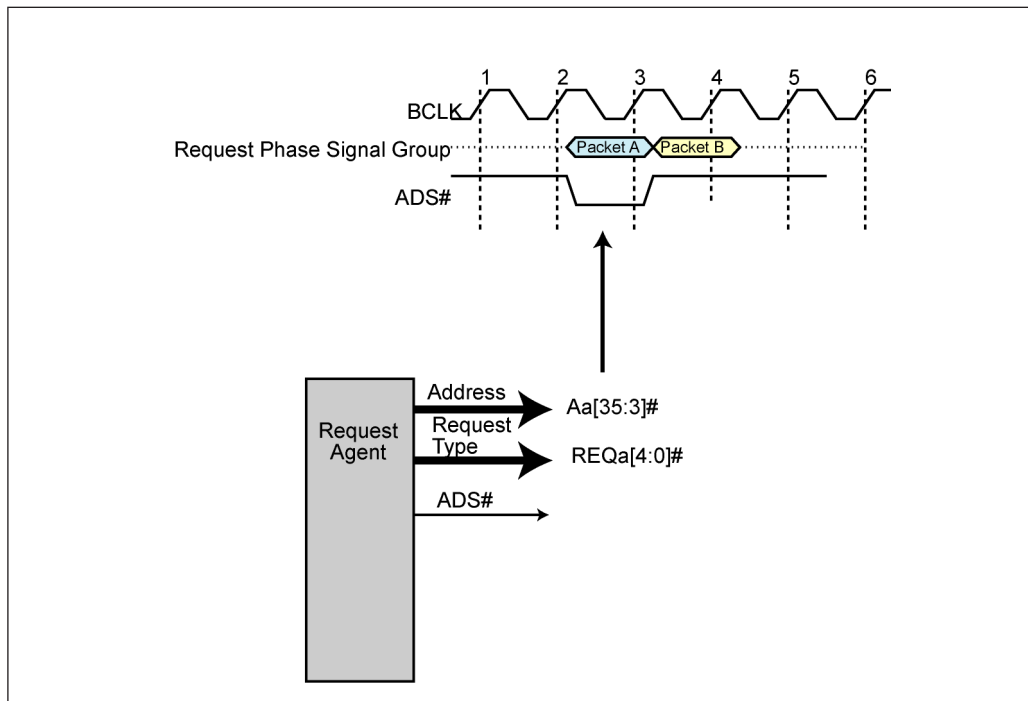
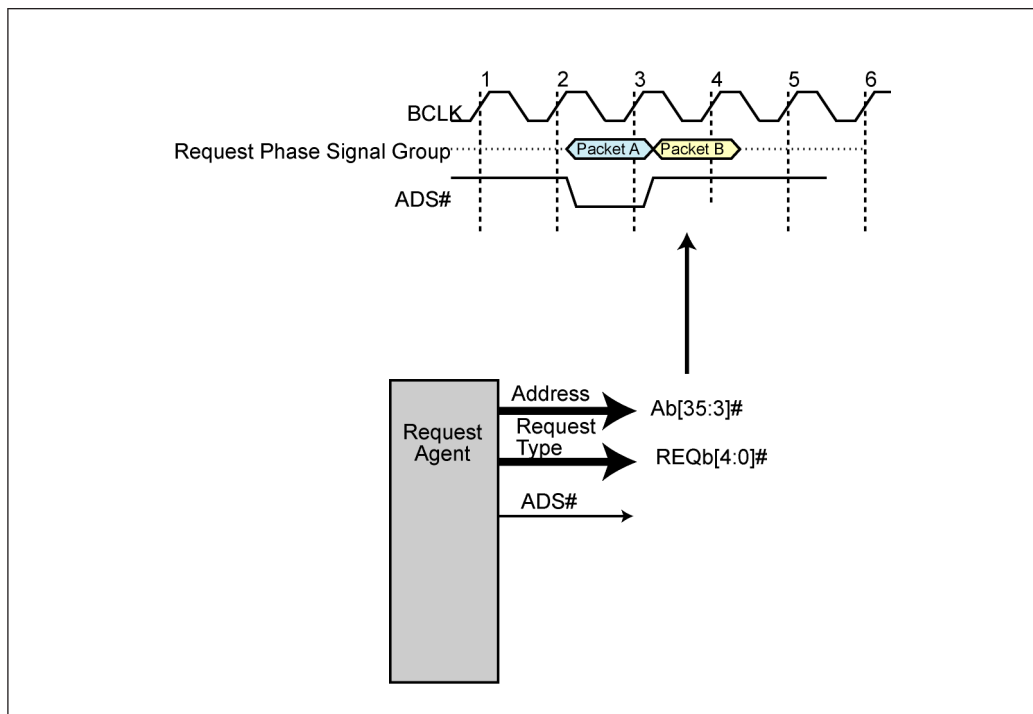


Figure 10-4: Packet B Signal Names



Introduction to the Transaction Types

The P6 FSB transaction types are listed in Table 10-2 on page 241. The table provides a brief description of each transaction type.

Chapter 10: P6 FSB Request & Error Phase Basics

Table 10-2: Currently-Defined Transaction Types

Transaction Type	Brief Description
Deferred Reply	Initiated by a Response Agent that had deferred the completion of an earlier transaction (because it was going to take a long time). This transaction addresses the Request Agent that had initiated the previously-deferred transaction and is used to deliver the transaction completion status (and, if a read, the read data) to the initiator. For more information, refer to “Pentium® 4 FSB Transaction Deferral” on page 1277 in the book.
Interrupt Acknowledge	Generated by a processor in response to an interrupt from an 8259 interrupt controller to read the interrupt vector. For more information, refer to “Interrupt Acknowledge Transaction” on page 273.
Special Transaction	Generated by the processor to broadcast a message regarding an internal event (e.g., Shutdown, Halt, etc.). For more information on the Special Transaction, refer to “P6 FSB Central Agent Transactions” on page 271.
Branch Trace Message	BTM. Generated by the processor when a branch is taken. Writes out the address of the branch instruction as well as the branch target address. For more information on the Branch Trace Message transaction, refer to “P6 FSB Central Agent Transactions” on page 271.
IO Read	Generated by the processor when executing an IN or INS instruction to read data or status from an IO device. Can also be generated by an agent other than a processor (e.g., a FSB-to-PCI bridge).
IO Write	Generated by the processor when executing an OUT or OUTS instruction to write data or a command to an IO device. Can also be generated by an agent other than a processor (e.g., a FSB-to-PCI bridge).

Table 10-2: Currently-Defined Transaction Types (Continued)

Transaction Type	Brief Description
Memory Read and Invalidate	Generated by the processor to gain exclusive ownership of a cache line. Caused by a cache read miss when performing a read / modify / write (RMW) operation, or before performing a RMW on a line in the cache in the shared state. Also generated when a write miss occurs in a WB memory area. For more information, refer to “Locking—Shared Resource Acquisition” on page 226.
Memory Code Read	Generated by the processor when fetching instructions from memory.
Memory Data Read	Generated by the processor when executing an instruction that requires the reading of data from memory (i.e., a load).
Memory Line WriteBack	Intel®’s official name for this transaction is “Memory Write (may not be retried)”, but the author chooses to call it the Memory Line WriteBack transaction (because that’s what it’s used for). It is generated by the processor when writing back a modified line to memory (a cast out) to make room for a new line being read into the cache.
Memory Write (may be retried)	Generated by the processor when executing an instruction that requires the writing of data to memory (i.e., a store).

Detailed Description of the P6 Request Phase

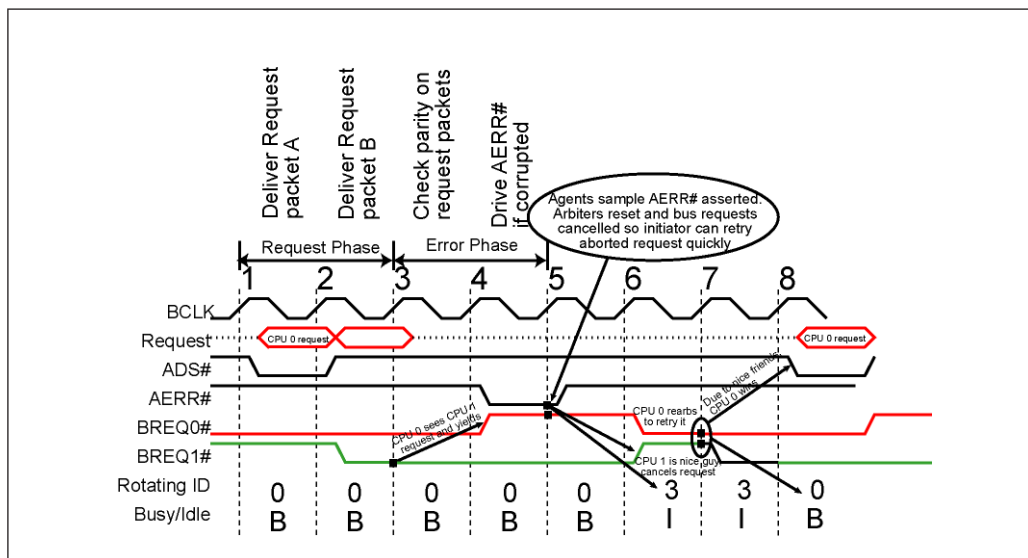
For a detailed description of the P6 processor’s Request Phase, refer to chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Introduction to the Error Phase

The Error Phase Purpose: Check for Good Receipt

As covered earlier, all FSB agents latch the two request packets during the first two BCLKs of the transaction (on the rising-edge of BCLKs 2 and 3 in Figure 10-5 on page 243). Three parity bits (AP0#, AP1# and RP#; they are not shown in the illustration) are latched along with packet A and again with packet B. During the third BCLK cycle of the transaction (the first clock of the Error Phase), all FSB agents check the parity of the two packets to determine if either or both were corrupted in flight. During the fourth clock of the transaction (the second and final clock of the Error Phase), any FSB agent that detected a parity error on either packet asserts the AERR# signal to the Request Agent. At the end of the Error Phase (on the rising-edge of BCLK 5 in the illustration), the Request Agent samples its AERR# input to determine if the transaction should proceed into its Snoop Phase or if it should be aborted. A more detailed explanation of the figure can be found in chapter 10 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition. A more detailed explanation of the P6 protocol's parity checking can be found in chapter 11 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Figure 10-5: Error Phase



Pentium® 4 FSB Protocol Eliminated the Error Phase

As mentioned at the beginning of this chapter, the Error Phase was eliminated with the advent of the Pentium® 4 processor. This does not mean that the FSB agents do not check and report bad parity, however. The AERR# signal was replaced by the MCERR# (Machine Check Error) signal and FSB agents that detected bad parity on either or both packets can assert MCERR# to report the corrupted Request Phase. A detailed description can be found in “Data Bus Parity” on page 1270 in the book and “The Request Phase Parity” on page 1204 in the book.

11

P6 FSB Snoop Phase Overview

Applicability to the Pentium® 4/Pentium® M

The description of the P6 FSB protocol's Snoop Phase is 100% compatible with the Pentium® 4 FSB protocol's Snoop Phase.

The Snoop Phase Has Two Purposes

In the Snoop Phase, the Request Agent samples the snoop result signals to determine:

1. If the currently-addressed Response Agent (i.e., the target) intends to complete the transaction now or it will be retried or its completion deferred until a later time (because it will take a while to complete).
2. If the transaction is a memory read or write that the Response Agent will complete now (i.e., it doesn't intend to defer its completion), does any other cache have a copy of the line and, if so, in what state (clean or modified) will its line be at the completion of the transaction?

The Snoop Phase Signals

Introduction

The signals that are sampled during the Snoop Phase are:

- **HIT#** and **HITM#**. These are the snoop result signals.
- **DEFER#**. This is not a snoop result signal. Rather, the currently-addressed target (i.e., the device acting as the transaction's Response Agent) will assert **DEFER#** in the transaction's Snoop Phase if it intends to issue a Retry or a Deferred respond when the transaction arrives at its Response Phase.

The Snoop Result Signals are Shared, **DEFER#** Isn't

There are three Snoop Phase signals (divided into two groups) that are sampled by the Request Agent and the Response Agent during the Snoop Phase:

1. The **HIT#** and **HITM#** are the signals used by the Snoop Agents (i.e., the internal processor caches) to deliver the cache snoop result, or to stall the completion of the Snoop Phase if one or more of the Snoop Agents isn't ready to deliver the snoop result. Both **HIT#** and **HITM#** might be driven by multiple snoopers if any of them need to stall the completion of the Snoop Phase until the snoop result is available. At that time, each of the snoopers that have been stalling would either drive neither (if it's a miss), just **HIT#** (if it's a hit on an E or S line), or just **HITM#** (if it's a hit on a modified line). **HIT#** and **HITM#** are shared, open-drain signals that may be driven by more than one device at a time.
2. Only the currently-addressed Response Agent (i.e., the target) is permitted to assert the **DEFER#** signal during the Snoop Phase. It therefore is not a shared, open-drain signal. The Response Agent will only assert **DEFER#** if it intends to issue a Retry or a Deferred response to the Request Agent.

The Agents Involved in the Snoop Phase

Refer to Figure 11-1 on page 248. The agents involved in the Snoop Phase of the transaction are the Request Agent, the Response Agent and the Snoop Agents. The following subsections introduce the roles of these devices during the transaction's Snoop Phase.

The Request Agent's Role in the Snoop Phase

The Request Agent issues the transaction request. This can be one of the processors or a Priority Agent (e.g., the North Bridge, aka the FSB-to-PCI Bridge). The Request Agent checks the snoop result presented in the Snoop Phase.

The Snoop Agent's Role in the Snoop Phase

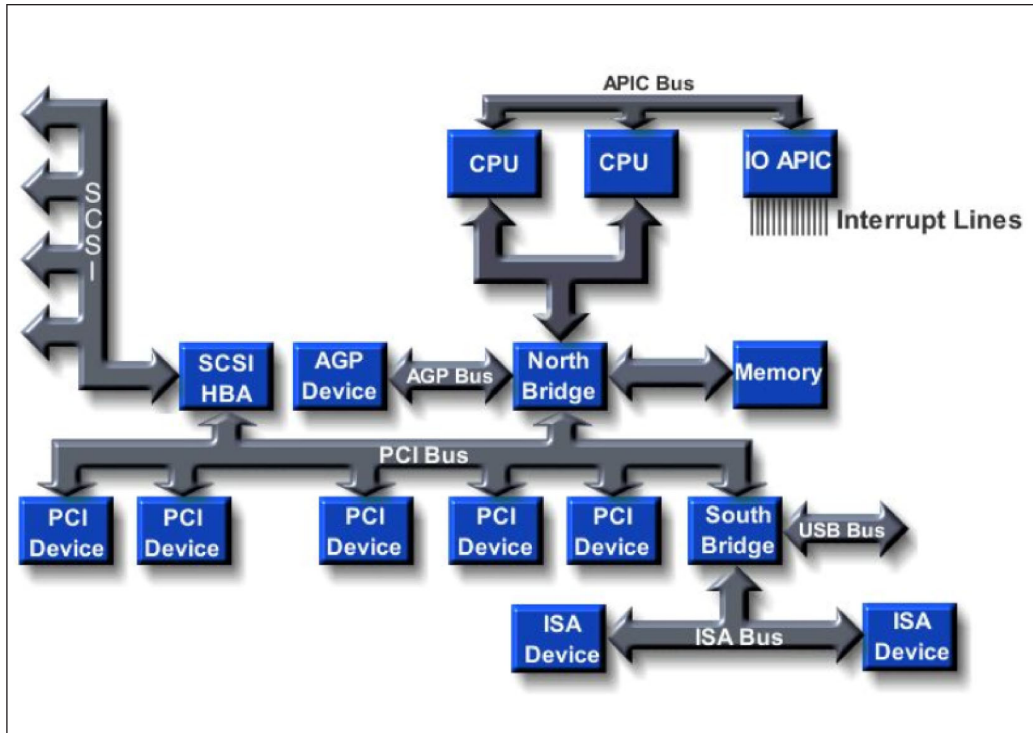
The Snoop Agents are the processors. They latch the transaction and, if it's a memory transaction, submit the memory address to all of their internal caches for a lookup. They present the snoop result to the Request Agent (on the HIT# and HITM# signals). If the snoop resulted in a hit on a modified line in a processor's cache, that Snoop Agent (i.e., a processor) supplies the modified line in the Data Phase (see the next section). If it's a non-memory transaction, the processors do not snoop the transaction or provide a snoop result.

The Response Agent's Role in the Snoop Phase

The Response Agent is the currently-addressed target. This could be system memory, a configuration register or an IO port within the North Bridge, a target residing on the PCI bus, or a target residing on another bus in the system (e.g., on the ISA bus in the illustration). If the Response Agent is system memory, it must observe the snoop response presented by the Snoop Agents:

- If the access results in a miss on all the caches, or in a hit on a clean line (i.e., the cached copy of the line is still the same as the line in system memory), the memory controller supplies the read data to the Request Agent or, if it's a write, accepts the write data presented by the Request Agent.
- If the transaction is a read that hits on a modified line:
 - The Snoop Agent supplies the modified line directly to the Request Agent in the Data Phase of the transaction.
 - In addition, when the system memory controller detects that it was a hit on a modified line (i.e., HITM# was asserted but HIT# wasn't), it latches the modified line as it's supplied to the Request Agent and updates the line in system memory.
- If the transaction is a write that hits on a modified line (i.e., HITM# was asserted but HIT# wasn't), the memory controller first accepts the write data from the Request Agent into one internal buffer, then accepts the modified line from the Snoop Agent into a second internal buffer, and finally merges the Request Agent's write data into the modified line and writes the resulting line into memory.

Figure 11-1: System Block Diagram



The Duration of the Snoop Phase Is Variable

The Snoop Phase begins immediately after the Error Phase completes and completes when a valid snoop result (something other than both HIT# and HITM# asserted) is presented to the Request and Response Agents by the Snoop Agents. Table 11-1 on page 249 defines the meaning of the various snoop results. Remember that although it may be asserted in a transaction's Snoop Phase, the DEFER# signal is not a snoop result signal and is never driven by a Snoop Agent. It is only driven by the transaction's Response Agent if it intends to issue a Retry or a Deferred response in the transaction's Response Phase.

Chapter 11: P6 FSB Snoop Phase Overview

Table 11-1: Snoop Result Table (0 = deasserted, 1 = asserted)

HIT#	HIT M#	DEFER#	Interpretation
0	0	0	Clean snoop (HIT# and HITM# both deasserted indicates a miss in all caches) and transaction completion will not be delayed by the issuance of a Retry or a Deferred response by the transaction's Response Agent.
0	0	1	The Response Agent has asserted DEFER#, indicating that it will issue either a Retry or a Deferred Response in the Response Phase. The Snoop Agents have indicated a miss in their caches: • If the Response Agent issues a Deferred response, the real snoop result will be delivered later in the Deferred Reply transaction initiated by the Response Agent. • If the Response Agent issues a Retry response, the Request Agent must retry the transaction from scratch.
0	1	0	The Response Agent did not assert DEFER#, so the transaction will continue. One of the Snoop Agents asserted HITM#, indicating a hit on a modified line. The Snoop Agent with the modified line will supply the line to the Response Agent (i.e., the system memory controller) in the Data Phase (and to the Request Agent if it's a read).
0	1	1	The Response Agent (i.e., the system memory controller) has asserted DEFER#, indicating that it will issue either a Retry or a Deferred Response in the Response Phase. One of the Snoop Agents asserted HITM# indicating a hit on a modified line. Assertion of HITM# in an unlocked transaction overrides the assertion of DEFER# (in other words, transaction completion will not be deferred). This situation occurs because the memory controller deferred so it can go read the line from memory (the memory access is slow; hence, the deferral), but the assertion of HITM# means the copy in memory is stale and the fresh copy will be supplied by the Snoop Agent that asserted HITM#. Upon seeing HITM# asserted, the memory controller cancels the read of the stale line from memory.

The Unabridged Pentium® 4

Table 11-1: Snoop Result Table (0 = deasserted, 1 = asserted) (Continued)

HIT#	HIT M#	DEFER R#	Interpretation
1	0	0	The Response Agent did not assert DEFER#, so the transaction will continue. The Snoop Agents have indicated a hit on a clean line (E or S state) in one or more caches.
1	0	1	The Response Agent (i.e., the system memory controller) has asserted DEFER#, indicating that it will issue either a Retry or a Deferred Response in the Response Phase. One or more of the Snoop Agents asserted HIT# indicating at one or more of the processors has a copy of the line in the clean state (E or S state). The Request Agent ignores the assertion of HIT#. In the Response Phase, the Response Agent will either issue a Retry or a Deferred response: • If the Response Agent issues a Retry, the Request Agent must retry the transaction from scratch again later. • If the Response Agent issues a Deferred response, the Response Agent initiates a Deferred Reply transaction at a later time and delivers the snoop result to the Request Agent. The Request Agent uses this deferred snoop result to choose the state of its copy of the line.
1	1	0	The Response Agent did not assert DEFER#, so the transaction will continue. HIT# and HITM# both asserted (a Snoop Stall) indicates that at least one Snoop Agent cannot deliver the snoop result yet. The Request Agent waits two clocks before sampling the snoop result again. The stall continues until a valid snoop result is delivered. Note that the state of DEFER# is ignored by the Request Agent until a good snoop result is delivered (something other than 11b on HIT# and HITM#).
1	1	1	Same as previous entry.

Chapter 11: P6 FSB Snoop Phase Overview

Is There a Snoop Stall Duration Limit?

No. The spec doesn't define a limit on how long the Snoop Phase may be stretched. However, to avoid hanging the system in the event of a failure, it is advisable for the chipset to monitor for excessive wait states inserted into the Snoop Phase and signal an error (e.g., via BERR#) if this condition occurs.

Memory Transaction Snooping

When a memory transaction is snooped, the snoop result affects the Request Agent, the Response Agent and the Snoop Agents. There are a number of possible cases. The type of memory transaction, whether or not other caches have a copy of the line, and the state of the line(s) in other caches all have an effect. In addition, sometimes the length of the data transfer has an effect. Table 11-2 on page 252 defines the scenarios and the results.

Table 11-2: Effects of Snoop

Transaction Snooped	Snoop Result	Effects of Snoop Result
Memory Read	Miss on all caches	The requested data is read from memory. If it's a Code Cache line fill, the line is placed in the Code Cache in the S state (the code cache is SI, not MESI) and in the E state in the L2 Cache. If it's a Data Cache line fill, the line is placed in the L2 and Data caches in the E state.
	Hit on E or S line in one or more caches	The data is read from memory. If it's a Code Cache line fill, the line is placed in the Code and L2 Caches in the S state. If it's a Data Cache line fill, the line is placed in the L2 and Data Caches in the S state. If a Snoop Agent had a copy of the line in the E state, it changes it to the S state.
	Hit on M line in one cache	The operation started as a read from memory, but this changes it to a read of the M line from the Snoop Agent's cache. From the memory controller's standpoint, the read has turned into an implicit writeback of the M line to memory. The memory controller issues an Implicit Writeback response to the Request Agent in the Response Phase and accepts the line from the Snoop Agent during the Data Phase. The Snoop Agent changes its copy from the M to the S state. The Request Agent may have requested less than a line, but a full line is returned in toggle mode order, critical qword first. The Request Agent takes the data it requested and ignores the rest. If it requested the whole line, the line is placed in the L2 and Code or Data caches in the S state.

Chapter 11: P6 FSB Snoop Phase Overview

Table 11-2: Effects of Snoop (Continued)

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory Write	Miss on all caches	The data is written to memory. There is no effect on the caches.
	Hit on E or S line in one or more caches	The Snoopers indicate a miss and invalidate their copies of the line (because the processors don't have the ability to snarf data being written to memory by other agents).
	Hit on M line in one cache	Case 1: writing less than a line. The memory controller accepts the write data from the Request Agent into an internal buffer and then accepts the modified line from the Snoop Agent's cache into another internal buffer. The memory controller then merges the write data into modified line and writes the updated line into memory. The Snoop Agent invalidates its copy of the line (M --> I).
		Case 2: writing a full line. The memory controller accepts the full line of write data from the Request Agent and then indicates its readiness to accept the modified line from the Snoop Agent's cache. The Snoop Agent can respond in one of two ways: • If it doesn't check the length of the write data, it asserts DBSY# and uses the data bus to supply the modified line to the memory controller. The memory controller then merges the write data into the modified line and writes the resultant line into memory. Alternately, the memory controller could discard the modified line and just use the write data to update the line in memory. • Alternatively, it could note that the Request Agent is writing a full line and decide that it won't send the modified line to the Response Agent (i.e., the memory controller). This is indicated by not asserting DBSY# in response to the assertion of TRDY#. The memory controller writes the line received from the Request Agent into memory. In either case, the Snoop Agent invalidates its copy of the line.

The Unabridged Pentium® 4

Table 11-2: Effects of Snoop (Continued)

Transaction Snoop	Snoop Result	Effects of Snoop Result
Memory Read and Invalidate for 32 bytes	Miss on all caches	The data is read from memory and is placed in the Data and L2 Caches. The transaction is a Read With Intent To Modify (RWITM). If the processor immediately modified one or more locations within the line (e.g., the update of a semaphore), the line is now in the M state. If the processor chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on E or S line in one or more caches	The Snoop Agents indicate a miss. The data is read from memory and is placed in the Data cache. The line(s) in the caches of other processors are invalidated. The transaction is a Read With Intent To Modify (RWITM). If the processor immediately modified one or more locations within the line (e.g., the update of a semaphore), the line is now in the M state. If the processor chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on M line in one cache	The data is supplied to the Request Agent and the Response Agent (i.e., the memory controller) from the modified line in the Snoop Agent's cache. The Snoop Agent invalidates its copy of the line. The Response Agent (i.e., the memory controller) may or may not write the line into memory. Since the line was read with the intent to modify it once it is placed in the Data Cache, the memory controller designer may choose not to waste memory bandwidth. If the processor immediately modified one or more locations within the line (e.g., the update of a semaphore), the line is now in the M state. If the processor chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.

Chapter 11: P6 FSB Snoop Phase Overview

Table 11-2: Effects of Snoop (Continued)

Transaction Snooped	Snoop Result	Effects of Snoop Result
Memory Read and Invalidate for 0 bytes	Miss on all caches	The processor would not perform this transaction.
	Hit on E or S line in one or more caches	• In the case where another processor has a copy in the E state, the processor performing the transaction would have had a miss on its cache and would be performing a Memory Read and Invalidate for a full line, not 0 bytes. • In the case where the processor performing the transaction has a copy in the S state, the transaction kills the line in the other processors that have it in the S state. The Snoopers indicate a miss. If the processor immediately modified one or more locations within the line (e.g., it updated a semaphore), the line is now in the M state. If the processor chooses not to update any locations in the line (e.g., the semaphore tested is already set), the line is placed in the E state.
	Hit on M line in one cache (this won't happen)	If another processor has a copy of the line in the M state, then the processor performing the transaction doesn't have a copy. It would therefore have a miss on its Data Cache and would perform a Memory Read and Invalidate for a full line, not 0 bytes.

Non-Memory Transactions Have a Snoop Phase

The protocol is consistent. All transactions, including non-memory transactions, have a Snoop Phase. For non-memory transactions, however, there are only three valid snoop responses:

1. A snoop miss (HIT# and HITM# both deasserted).
2. Defer (DEFER# asserted), indicating that the currently-addressed target (i.e., the system memory controller) will issue a Retry or a Deferred response in the transaction's Response Phase.

3. A Snoop Stall (HIT# and HITM# both asserted), indicating that the Snoop Phase is to be extended by two clocks. A non-memory Response Agent is permitted to stretch the snoop response to give itself more time for internal operations to complete before presenting a miss or a DEFER# asserted as the snoop result.

The non-memory transaction types are the Deferred Reply, Interrupt Acknowledge, Special, Branch Trace Message, IO Read, and IO Write transactions. For more information on these transaction types, refer to the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition. In addition, they are described in the following chapters of this book:

- “Pentium® 4 FSB Transaction Deferral” on page 1277 in the book.
- “Pentium® 4 FSB Central Agent Transactions” on page 1301 in the book.
- “Pentium® 4 FSB IO Transactions” on page 1295 in the book.

12 *P6 FSB Response & Data Phase Basics*

Applicability to the Pentium® 4/Pentium® M

The Response Phase

The description of the P6 FSB protocol's Response Phase is 100% compatible with the Pentium® 4 FSB protocol's Response Phase.

The Data Phase

The Pentium® 4 FSB protocol's Data Phase is implemented differently than it was for the P6 processor. The two protocols have the following similarities and differences:

- The following Data Phase signals are used identically in both protocols:
 - **DBSY#**. **Data Bus Busy**. Asserted by the agent that will drive data on the data bus (the Response Agent on a read; the Request Agent on a write; the Snoop Agent on a snoop hit on modified line) when it takes ownership of the data bus. **DBSY#** stays asserted at least until one clock before the final transfer completes.

The Unabridged Pentium® 4

- DRDY#. **Data Ready**. Asserted by the agent driving data on the data bus to indicate that the data is valid. DRDY# can be deasserted during the Data Phase to insert wait states into the Data Phase.
- D[63:0]#. **Data Bus**. The eight, one-byte data paths that are used to transfer up to a qword at a time.
- The following are differences between the two protocols:
 - While the P6 data bus could be either ECC- or parity-protected, the Pentium® 4 data bus is parity protected. In addition, the Pentium® 4 implements a different data bus parity protection scheme.
 - While a qword is transferred on the rising-edge of each BCLK cycle on the P6 FSB, a set of data strobe signals are used to transfer up to four qwords per BCLK cycle on the Pentium® 4 FSB.
 - Data Bus Inversion is used to saved on power consumption on the Pentium® 4 FSB.

Response Phase Basics

The Response Phase Start Point

The Response Phase starts immediately after the Snoop Phase completes (i.e., when a valid snoop result is received).

The Purpose of the Response Phase

In the Response Phase of the transaction, the Response Agent must indicate one of the following to the Request Agent:

- It will service the request immediately.
- It can't service it now, but will be able to later.
- It will service it off-line and get back to the Request Agent with the completion later.
- It is broken and can't service the request at all.
- A Snoop Agent has a copy of the targeted line in the modified state and will supply the line to the memory controller (and to the Request Agent if it's a read).

The possible responses are:

- **Retry** response.
- **Deferred** response.
- **Hard Failure** response.

Chapter 12: P6 FSB Response & Data Phase Basics

- **No Data** response.
- **Implicit Writeback** response.
- **Normal Data** response.

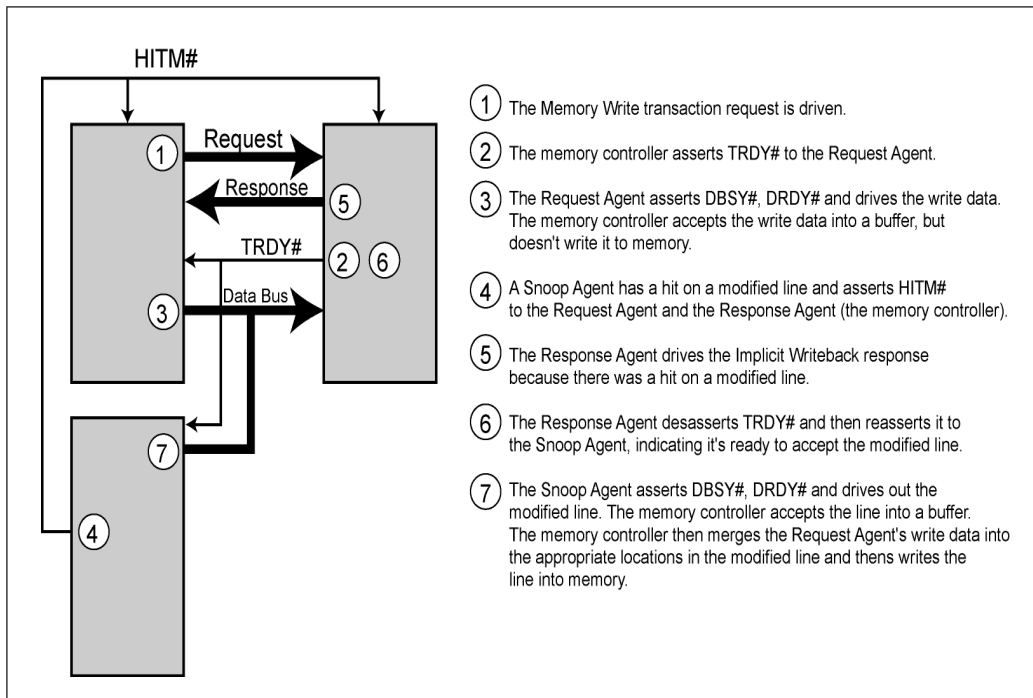
The Response Phase Signal Group

The following signals are used in the Response Phase:

- **RS[2:0]#. Response bus.** Used to deliver the response to the Request Agent.
- **RSP#. Response bus parity bit.** The parity signal that covers RS[2:0]#. It is an even parity signal that, if necessary, is driven low to force an even number of electrical lows in the overall 4-bit pattern that includes RSP#.
- **TRDY#. Target Ready.** Only asserted by the Response Agent if data is to be written to the Response Agent by either the Request Agent, a Snoop Agent (for an implicit writeback of a modified line), or both. The assertion of TRDY# indicates the Response Agent's readiness to accept the write data.

Figure 12-1 on page 259 illustrates the series of actions taken when a Memory Write results in a hit on a modified line in a processor cache.

Figure 12-1: Actions Taken on a Memory Write That Hits on a Modified Line



The Response Phase End Point

The Response Phase ends when the Response Agent delivers a valid response to the Request Agent. This implies that the Response Agent can stall the Response Phase (i.e., insert wait states) until it is ready to present its response.

The spec doesn't place a limit on the number of wait states that may be inserted into the Response Phase of a transaction. However, the system designer may choose to monitor the behavior of agents to ensure that none of them inserts excessive wait states. This would adversely affect all subsequently-issued transactions that are awaiting delivery of their respective response.

The List of Responses

Table 12-1 on page 260 lists the possible responses that can be presented on RS[2:0]#.

Table 12-1: Response List (0 = deasserted, 1 asserted)

RS[2:0]#	Description
000b	Idle. This is referred to as the idle state. None of the response signals are asserted. This is the state of RS[2:0]# before and after the response has been delivered to the Request Agent. In other words, immediately upon entry into the Response Phase, RS[2:0]# are in this state and will remain in this state until a valid response is presented. When any of the valid responses are driven (for one clock), one or more of the RS[2:0]# signals are driven low. A look at the other table entries shows that all of the valid response patterns have at least one of the RS signals asserted (1 = asserted—an electrical low). After one clock, the response is removed. The RS signals then all return to the deasserted state (in other words, back to the idle state).
001b	Retry. The Response Agent may command the Request Agent to Retry the transaction repeatedly (due to a temporary logic busy condition) until it succeeds (or fails). In other words, it can't service the request now, but will be able to later. An example would be if a Request Agent started a Memory Write when the target device's (Response Agent's) Posted Memory Write Buffer was full.

Chapter 12: P6 FSB Response & Data Phase Basics

Table 12-1: Response List (0 = deasserted, 1 asserted) (Continued)

RS[2:0]#	Description
010b	Deferred. The Response Agent may inform the Request Agent that due to the amount of time that it will take to perform the requested access, the Response Agent has memorized the transaction. It issues a Deferred response to tell the Request Agent to end the transaction without any data being transferred. The Response Agent then performs the requested access off-line from the FSB and, when it has completed the access, arbitrates for ownership of the FSB and performs a Deferred Reply transaction to deliver a completion notice to the Request Agent.
011b	Reserved.
100b	Hard Failure. The Response Agent may indicate a Hard Failure to the Request Agent. In other words, it is broken and can't service the request at all.
101b	No Data. If the transaction is one that doesn't require the Response Agent to send data to the Request Agent (i.e., it is a write transaction, a Special transaction, a Memory Read, or a Memory Read and Invalidate transaction for 0 bytes), the Response Agent indicates that as requested, No Data will be returned to the Request Agent. The request will be service immediately (i.e. it will not be retried or deferred).
110b	Implicit Writeback. This is the response given if the memory transaction resulted in a hit on a modified line (i.e., HITM# was asserted in the Snoop Phase). This means that the Snoop Agent that has the modified line will supply the modified line to the Response Agent (i.e., the memory controller) as well as to the Request Agent (if it's a read transaction). The Request Agent may be attempting to read less than a line of information and, if the Snoop Agent supplies the data, it always sends the full line. The Implicit Writeback response tells the Request Agent that four qwords (32 bytes) will be transferred rather than the smaller data packet actually requested. The four qwords are transferred in toggle mode order, critical qword first. This means that the first qword sent back by the Snoop Agent will be the first one requested by the Request Agent and, if a second qword was also requested (i.e., a 16 byte read request), the second qword sent back is the first qword's toggle mode partner. The Request Agent should just take the qword(s) requested and ignore the rest. The memory controller (i.e., the Response Agent), on the other hand, will accept the full line.

Table 12-1: Response List (0 = deasserted, 1 asserted) (Continued)

RS[2:0]#	Description
111b	Normal Data. If the transaction is any form of a read transaction (Memory Read, Memory Read and Invalidate for 32 bytes, IO Read, or Interrupt Acknowledge), the Response Agent indicates whether or not it will return the requested data in the Data Phase (it may choose to defer delivery of the read data until a later time). To indicate that the data will be delivered, it issues the Normal Data response.

The Response Phase May Complete a Transaction

In any transaction that does not involve a data transfer, the delivery of the response ends the transaction. This would include the following transactions:

- The Special transaction broadcasts a processor message encoded on the Byte Enables, but does not read or write data.
- A 0-byte Memory Read, a Memory Read and Invalidate for 0 bytes, or an IO Read for 0 bytes transfers no data.
- A transaction that receives a Retry response.
- A transaction that receives a Hard Failure response.
- A read transaction that receives a Deferred response from the Response Agent. The Response Agent will initiate a Deferred Reply transaction at a later time and transfer the read data to the Request Agent at that time.

Any transaction that performs a read or write that does not receive a Deferred, Retry, or Hard Fail response will have both a response and a Data Phase.

Detailed Description of the Response Phase

A detailed description of the P6 Response Phase may be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

Data Phase Basics

The P6 Data Phase Signal Group

The P6 Data Phase signal group consists of the following:

- **DBSY#. Data Bus Busy.** Asserted by the agent that will drive data on the data bus (the Response Agent on a read; the Request Agent on a write; the Snoop Agent on a snoop hit on modified line) when it takes ownership of the data bus. DBSY# stays asserted at least until one clock before the final transfer completes.
- **DRDY#. Data Ready.** Asserted by the agent driving data on the data bus to indicate that the data is valid. DRDY# can be deasserted during the Data Phase to insert wait states into the Data Phase.
- **D[63:0]#. Data bus.** The eight, one-byte data paths that are used to transfer up to a qword at a time. The eight data paths are:
 - D[7:0]#. Data path 0. This data path is used to transfer a byte to or from location 0 in the current qword.
 - D[15:8]#. Data path 1. This data path is used to transfer a byte to or from location 1 in the current qword.
 - D[23:16]#. Data path 2. This data path is used to transfer a byte to or from location 2 in the current qword.
 - D[31:24]#. Data path 3. This data path is used to transfer a byte to or from location 3 in the current qword.
 - D[39:32]#. Data path 4. This data path is used to transfer a byte to or from location 4 in the current qword.
 - D[47:40]#. Data path 5. This data path is used to transfer a byte to or from location 5 in the current qword.
 - D[55:48]#. Data path 6. This data path is used to transfer a byte to or from location 6 in the current qword.
 - D[63:56]#. Data path 7. This data path is used to transfer a byte to or from location 7 in the current qword.
- **DEP[7:0]#. Data Bus ECC/Parity.** DEP[0]# is the ECC or parity bit associated with the byte being transferred over data path 0, DEP[1]# is the ECC or parity bit associated with the byte being transferred over data path 1, etc.

Detailed Description of the Data Phase

A detailed description of the P6 Data Phase may be found in chapter 13 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*.

13 *P6 FSB Deferral Basics*

Applicability to the Pentium® 4/Pentium® M

The description of transaction deferral in this chapter also applies to the Pentium® 4/Pentium® M FSB protocol.

Permission to Defer Transaction Completion

The Request Agent uses the state of its DEN# output (Defer Enable) in packet B of the Request Phase to grant or deny the Response Agent permission to defer the transaction's completion until a later time. Note that this does not deny the Response Agent permission to retry the transaction:

- When DEN# is deasserted, permission to defer is denied. Permission is denied under the following circumstances:
 - When a processor is performing a memory write to cast a modified line back to memory to make room for a new line being read into the Data Cache.
 - When an agent is performing a locked transaction series.
 - When an agent is performing a Deferred Reply transaction.
 - A Request Agent that requires fast completion of its transaction may deassert DEN#, denying the Response Agent permission to defer the transaction for later completion.
- When DEN# is asserted, permission is granted to assert DEFER# and issue a Deferred response.

DEFER# Assertion Delays Transaction Completion

When the currently-addressed Response Agent does not assert DEFER# in the Snoop Phase, the agent is promising to complete the transaction now (with something other than a Retry or a Deferral for later completion).

If the Response Agent asserts DEFER# in the Snoop Phase, the Response Agent will either issue a Retry or a Deferred response to the Request Agent in the Response Phase. In both of these cases, the Response Agent is indicating that it cannot complete the transaction now. There are two possible cases:

1. If the Request Agent must complete this transaction before it can proceed with subsequent transactions (in other words, it's important for proper device driver operation that its transactions complete in program-initiated order), then the Request Agent must not issue any new transactions until it successfully completes the current transaction at a later time (by reattempting the retried transaction or by receiving the corresponding Deferred Reply transaction from the Response Agent).
2. If the Request Agent doesn't have to complete this transaction before issuing subsequent transactions, it can choose to issue different transactions before reattempting the transaction that received a Retry or before receiving the corresponding Deferred Reply transaction from the Response Agent.

Mail Delivery Analogy

A good analogy for a deferred transaction is sending a letter with a request for a delivery completion notification. The read or write transaction is delivered to the Response Agent who memorizes it as well as the sender's address. At a later time, the Response Agent sends a delivery notice back to the transaction originator (and, if it's a read, delivers the requested data). Note that the Response Agent may have had a problem when delivering the transaction to the addressed target and must then report the problem faithfully in the Response Phase of the Deferred Reply transaction.

Example System Operation Overview

Refer to the system model pictured in Figure 13-1 on page 269 during the following discussion. The illustration shows a cluster of two processors. The processors share access to system memory.

Chapter 13: P6 FSB Deferral Basics

Any processor may issue a transaction request targeting a device residing beyond the Root Complex. During the startup process, the Root Complex is configured to recognize the IO and memory ranges assigned to the devices that reside beneath it.

When any of the processors attempt to perform a read or write with a target residing beneath the Root Complex, it can result in very long latency during the Data Phase of the transaction.

The Wrong Way: Stall Until You Have Completion

In Figure 13-1 on page 269, assume that a processor initiates an IO or a memory-mapped IO read from a register within the IEEE 1394 FireWire controller. This device resides on the PCI bus at the bottom of hierarchy. If the Root Complex handled the read by keeping the FSB Data Bus busy (i.e., keeping DBSY# asserted until the requested read data is finally returned by the FireWire controller), the Data Bus portion of the FSB will be tied up for an extensive period of time. This will cause any subsequently issued FSB transactions to stall when they reach their Data Phases.

The Right Way: Break the Connection and Reply Later

Rather than tie up the FSB, the Root Complex asserts DEFER# to the processor in the Snoop Phase and issues the Deferred response in its Response Phase. This informs the processor that the transaction will be completed at a later time. The processor's FSB interface moves the transaction from its IOQ to its Deferred Transaction Queue. All other FSB agents delete the transaction from their IOQs and cease to track it. The Root Complex has "memorized" the following information:

- Whether or not the Response Agent is permitted to defer the transaction (refer to "Permission to Defer Transaction Completion" on page 265).
- The address and transaction type.
- The DID (Deferred ID) field delivered in packet B of the Request Phase. This will permit the bridge to "address" the processor later when it initiates its Deferred Reply transaction to complete the original transaction.
- If the transaction is a read, the amount of data to be read.
- If the transaction is a write, the data to be written.

When the read or write transaction eventually completes on the PCI bus, the Root Complex must deliver completion notification to the processor that initiated the read or write. To do this, it arbitrates for Request Phase signal group ownership (by asserting BPRI#). When it has acquired ownership of the Request Phase signal group, it initiates a Deferred Reply transaction. The DID that was originally received from the processor is used as the address in request packet A. All bus agents latch the transaction request and detect that it is a completion notice for a previously-issued transaction that is still awaiting completion (i.e., it is a Deferred Reply transaction). The processor that originally initiated the transaction has a match on its agent ID and transaction ID, identifying this as the completion notice for the previously-deferred transaction.

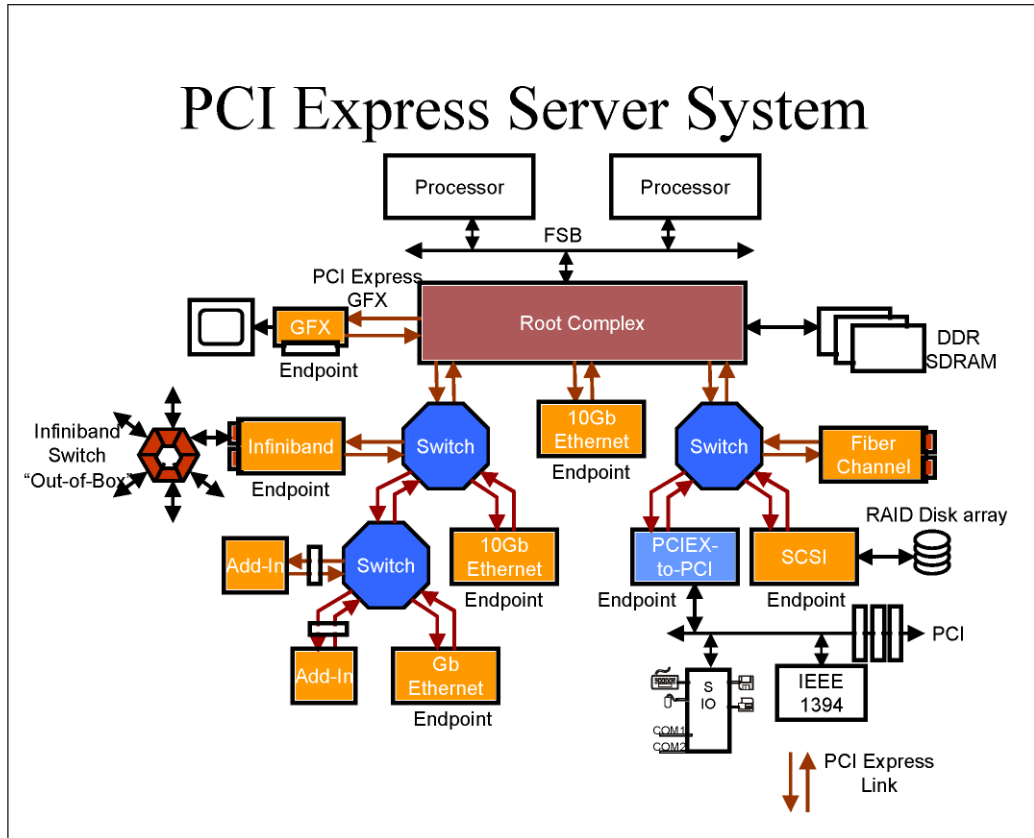
Root Complex Should be a Faithful Messenger

During the Snoop Phase of the Deferred Reply transaction, the Root Complex and the processor change roles—the processor becomes the Request Agent and the Root Complex becomes the Snoop Agent. *The Deferred Reply transaction is never snooped by snoop agents.* Rather, the Root Complex now supplies the snoop result that was received on the PCI bus (since there aren't any caches on PCI buses, the result is a miss—HIT# and HITM# are deasserted).

In the Response Phase, the Root Complex acts as the Response Agent and delivers the response it received from the PCI target:

- If the transaction was a write and the data was delivered to the PCI target successfully, the response is the **No Data** response.
- If the transaction was a write and the data wasn't delivered to the PCI target successfully (due to a Master Abort, Target Abort, or a parity error), the response is the **Hard Failure** response indicating that the transaction failed.
- If the transaction was a read and the data was read from the PCI target successfully, the response is the **Normal Data** response and the read data is supplied in the Data Phase.
- If the transaction was a read and the data wasn't read successfully, the response is the **Hard Failure** response and the transaction failed.

Figure 13-1: System with Four Processors and Two FSB/PCI Bridges



What If HITM# and DEFER# Are Both Asserted?

For all transactions that are not part of a locked transaction series (i.e., LOCK# is not asserted), the assertion of HITM# overrides the assertion of DEFER#. In other words, if HITM# is asserted, the Response Agent provides the Implicit Writeback response in the Response Phase and the Snoop Agent writes the modified line to memory (and, if it's a read transaction, the line is also received by the Request Agent).

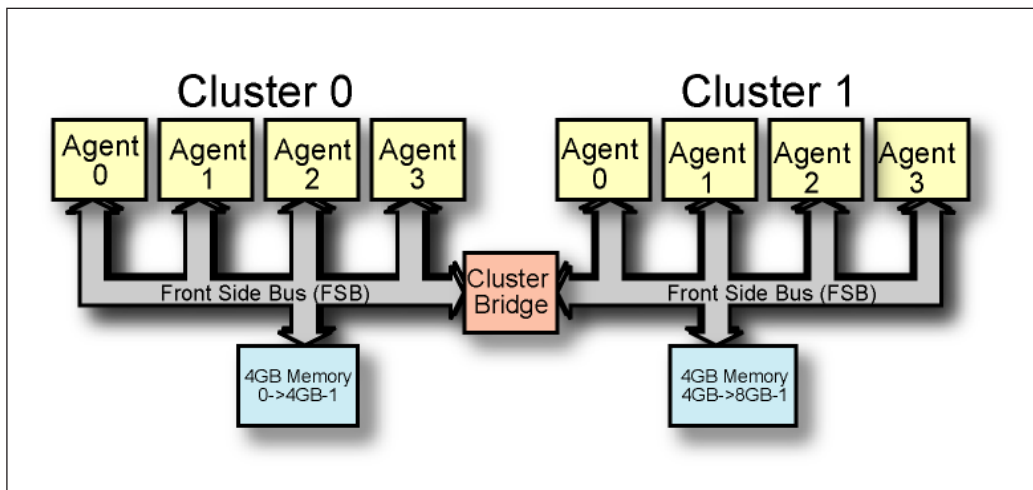
Consider a system with two processor clusters linked by a cluster bridge (see Figure 13-2 on page 270). A processor on the cluster 0 FSB initiates a memory read from the cluster 1 memory controller and the transaction is latched by all of the processors on the cluster 0 FSB as well as by the Cluster Bridge. The Cluster

The Unabridged Pentium® 4

Bridge looks at the memory address, determines that it must pass the transaction onto the Cluster 1 FSB. Since this will take some time, the Cluster Bridge asserts DEFER# in the Cluster 0 transaction to indicate that it will issue a Deferred response in the transaction's Response Phase.

However, the processors in Cluster 0 snoop the address in their caches and it results in a hit on a modified line in the cache of one of the cluster 0 processors. In other words, the cluster 0 processor with the modified copy of the line had read the line from the cluster 1 memory controller earlier in time and had subsequently made one or more changes to the line in its Data Cache. In this case, the most up-to-date copy of the line is the modified one in the cache. Rather than passing the read to the cluster 1 memory controller, the Cluster bridge cancels its intended Deferred Response. The processor that experienced the hit on a modified copy of the line therefore supplies the requested read data to the Request Agent in Cluster 0. In addition, the Cluster Bridge also latches the modified line, arbitrates for ownership of the cluster 1 FSB and writes the fresh copy of the line to the cluster 1 memory controller.

Figure 13-2: Two Processor Clusters Linked by Cluster Bridge



Detailed Deferred Transaction Description

For a more detailed description of P6 deferred transactions, refer to chapter 14 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition.

14 *P6 FSB Central Agent Transactions*

Applicability to the Pentium® 4/Pentium® M

Except as noted, everything in this chapter also applies to the Pentium® 4/Pentium® M processors as well.

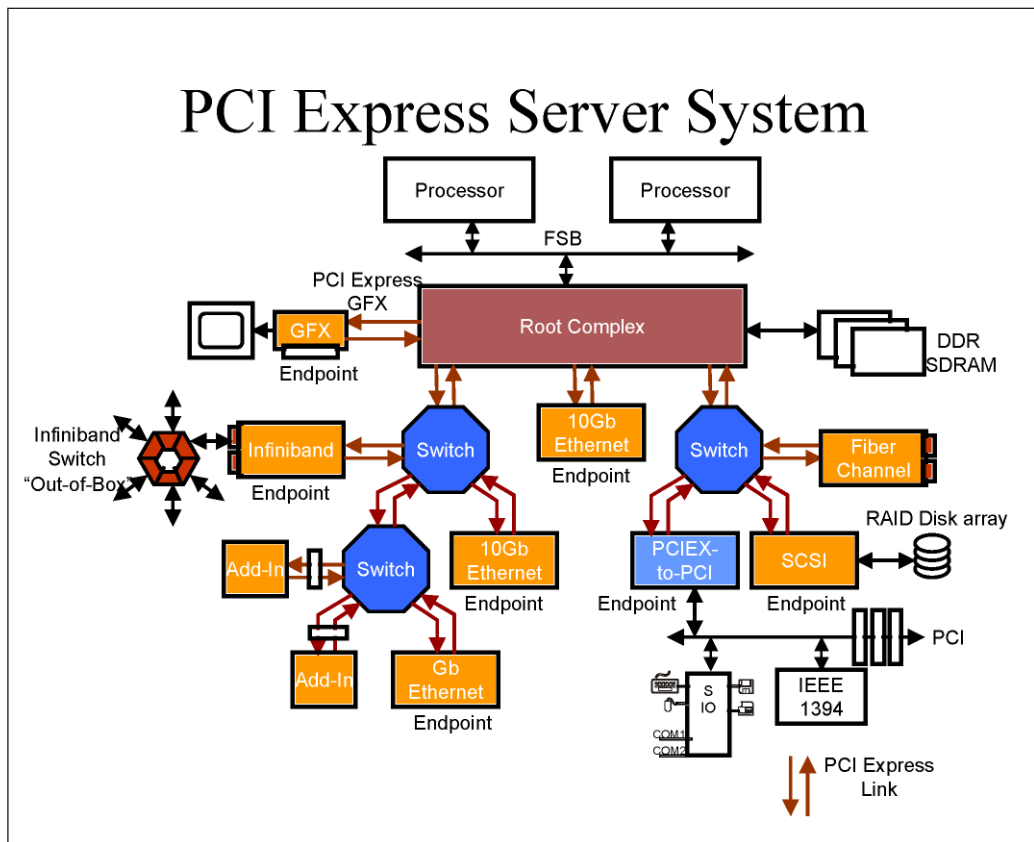
Point-to-Point vs. Broadcast

Most transactions are point-to-point transactions—the Request Agent addresses a specific area of memory or IO space for a read or a write and the addressed target acts as the transaction’s Response Agent.

Some transactions generated by the processor don’t target any specific memory or IO device, however, and some device on the FSB must act as the transaction’s Response Agent. In the spec, this device is typically referred to as the Central Agent and the device that typically fulfills this role is the bridge that connects the FSB to the platform’s fabric (e.g., see the Root Complex in Figure 14-1 on page 272 and the FSB-to-PCI Bridge in Figure 14-2 on page 274). Rather, the processor is performing one of the following operations:

1. An **Interrupt Acknowledge** transaction to request the interrupt vector from the 8259A Interrupt Controller. In this case, the bridge that connects the FSB to the platform fabric acts as the Response Agent (because the Interrupt Controller typically resides within or beyond the bridge).
2. A **Special** transaction to broadcast a message to the platform. No one is targeted by the transaction, but nonetheless, someone has to act as the Response Agent (once again, this device is typically the bridge that connects the FSB to the platform fabric).
3. A **Branch Trace Message (BTM)** transaction to inform a debug tool that, upon execution of a conditional branch instruction, the condition was met and the branch was taken. Once again, no one in particular is being addressed and yet someone has to act as the Response Agent (typically the the bridge that connects the FSB to the platform fabric).

Figure 14-1: Example PCI Express System



No Address Is Output

Since the Central Agent transactions do not target any specific device, the contents of the Address Bus in packet A of the Request Phase is reserved and has no meaning. The implicit target of the transaction is the device that fills the role of the Central Agent (i.e., the chipset).

Interrupt Acknowledge Transaction

Background

An IA32-based system frequently incorporates an Interrupt Controller that receives interrupt requests from IO devices and passes them to the processor (or, in a multiprocessor system, to the processor cluster). The Interrupt Controller will either consist of a pair of cascaded 8259As (as in a single processor system), or an IO APIC module in a multiprocessor system. Refer to Figure 14-2 on page 274. In older chipsets, the Interrupt Controller was typically incorporated in the PCI-to-ISA Bridge (commonly referred to as the South Bridge), or in the ICH (IO Control Hub) in later chipsets. This was a strategically convenient place for it because the interrupt requests from PCI and ISA targets could easily be connected to it.

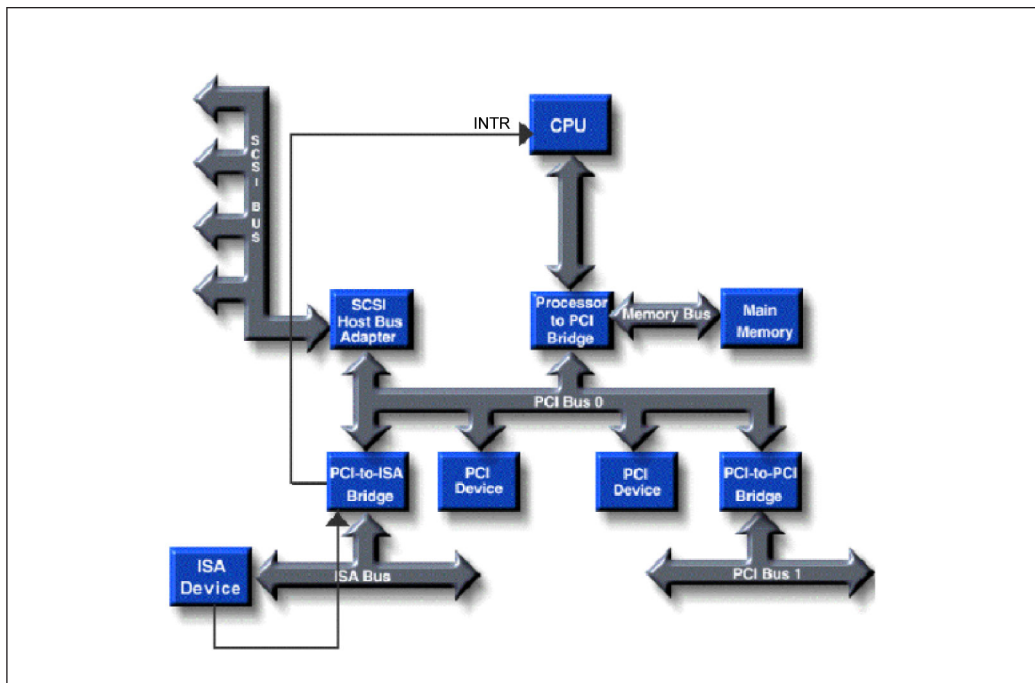
Assuming that the system uses the 8259A Interrupt Controller, the Interrupt Controller asserts its INTR (Interrupt Request) output when it detects any interrupt requests from IO devices (see Figure 14-2 on page 274). This is connected to the INTR pin (also referred to as the LINT0 pin) on one of the processors (typically the only processor). In response to its assertion, the processor takes the following actions:

1. Assuming that recognition of external interrupts is enabled (in other words, the programmer has not executed a CLI instruction), the processor will recognize the request when it completes the execution of the current instruction.
2. The processor temporarily ceases execution of the interrupted program.
3. The processor generates an Interrupt Acknowledge transaction on the FSB to read the interrupt vector of the highest priority request from the Interrupt Controller.
4. The North Bridge (i.e., the bridge between the FSB and the first PCI bus) passes the transaction to the PCI bus to make it visible to the chip that contains the Interrupt Controller (i.e., the South Bridge in the example system).

The Unabridged Pentium® 4

5. The Interrupt Controller supplies the 8-bit interrupt vector to the FSB-to-PCI Bridge.
6. The North Bridge supplies the interrupt vector to the processor.
7. The processor uses the 8-bit vector as an index into the Interrupt Table in memory and reads the CS:EIP value from the selected entry.
8. The processor pushes the contents of its CS, EIP and EFlags registers into stack memory (to mark its place in the interrupted program).
9. The processor then disables recognition of additional external interrupts (i.e., it clears EFlags[IF] to 0).
10. Using the new CS:EIP value, the processor starts fetching the instructions that comprise the target interrupt service routine and executes them.

Figure 14-2: An External, Hardware Interrupt Delivered to the Processor's INTR Pin



How the P6 Processors Differed from Earlier IA32 Processors

The P6 processors handled the Interrupt Acknowledge differently than earlier IA32 processors. The earlier processors generated two, back-to-back Interrupt Acknowledge transactions: one to command the Interrupt Controller to prioritize its pending requests, the second to request the interrupt vector associated with the highest priority request. However, the P6 processor only generated one Interrupt Acknowledge transaction.

Special Transaction

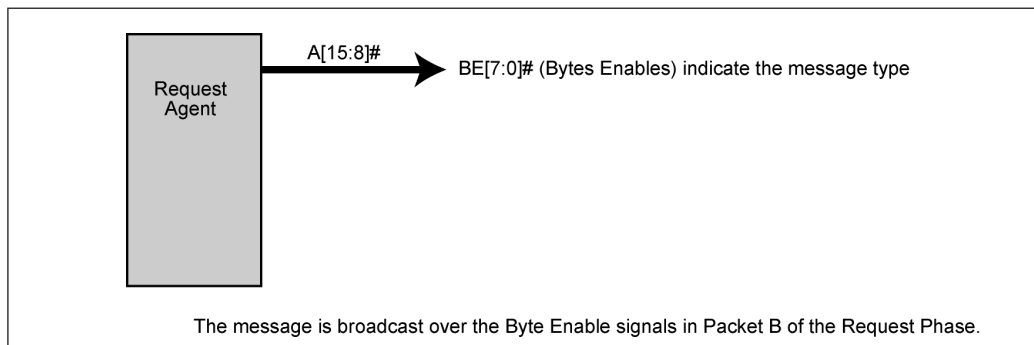
General

Under special circumstances, an IA32 processor generates a Special transaction to broadcast a message to the platform. In other words, this is not a point-to-point transaction that targets a particular agent. As indicated earlier, every transaction requires that a Response Agent respond, and the Central Agent is the Response Agent for the Special transaction. As stated earlier, the address is reserved and has no meaning and the implicit target of the Special transaction is the Central Agent.

The Message Is Encoded on the Byte Enables

Refer to Figure 14-3 on page 275. The message being transmitted by the processor is presented on its Byte Enable outputs during the transmission of packet B in the Request Phase.

Figure 14-3: The Message Is Encoded on the Processor's Byte Enables



The Message Types

As stated in the previous section, the message type is driven out on BE[7:0]# (A[15:8]#) in packet B. Table 14-1 on page 276 indicates the messages defined for the P6 processor family.

Table 14-1: Message Types

BE[7:0]#	Message Type
00h	Nop. This message was reserved for the P6 processor family, but it is used in the Pentium® 4/Pentium® M processors [see “Boot Strap Processor (BSP) Selection” on page 885 in the book for more information].
01h	Shutdown. Indicates that the processor incurred a severe software error. As with previous IA32 processors, the P6 generates this message when it encounters a triple-fault condition. In other words, it has received another exception while attempting to call the Double Fault exception handler [see “Double Fault Exception (8)” on page 301 in the book for more information]. In response to the triple-fault, the processor ceases program execution and generates this message. Whether or not a system pays any attention to this message and, if so, the action taken by the system, is system design-specific. In a PC system, the chipset asserts reset to the system and then removes it. This causes the processor to re-execute the POST and reboot the OS.
02h	Flush. Generated by the processor when it executes an INVD (invalidate caches) instruction . This instruction causes the processor to invalidate all of its internal caches without writing modified lines back to memory. The processor also broadcasts this message to inform an external cache (if present) that it should also invalidate its contents. It should be noted that this message does not cause other processors to invalidate their internal caches. If this is the programmer’s intent, the processor’s Local APIC should be instructed to send an IPI (Inter Processor Interrupt message packet) to the other processors over the APIC bus instructing them to do the same.
03h	Halt. Generated by the processor when it executes a HLT instruction . Whether or not a system pays any attention to this message and, if so, the action taken by the system, is system design-specific. In a PCI-based system, the chipset generates a PCI Special Cycle transaction and broadcasts the Halt message in the Data Phase of the PCI transaction (in case any of the PCI agents care that a processor has halted). In a laptop system, the chipset may power down some subsystems when the processor has halted.

Chapter 14: P6 FSB Central Agent Transactions

Table 14-1: Message Types (Continued)

BE[7:0]#	Message Type
04h	Sync. Generated by the processor when it executes an WBINVD (Write Back and Invalidate caches) instruction . This instruction causes the processor to first write back all modified lines to memory, after which it invalidates all of its internal caches. The processor then broadcasts this message to inform an external cache (if present) that it should write all of its modified lines to system memory and then invalidate all of its line copies. It should be noted that this message does not cause other processors to take this action in their internal caches. If this is the programmer's intent, the processor's Local APIC should be instructed to send an IPI (Inter Processor Interrupt message packet) to the other processors over the APIC bus instructing them to do the same.
05h	Flush Acknowledge. When the chipset asserts the processor's FLUSH# input, this causes the processor to first write back all modified lines to memory, after which it invalidates all of its internal caches. The processor then broadcasts this message to inform the chipset that it has completed the operation.
06h	Stop Grant Acknowledge. When the chipset asserts the processor's STPCLK# (Stop Clock) input, the processor turns off the clock to all of its internal units with the exception of its FSB interface, the Time Stamp Counter (TSC), and the Local APIC. This is referred to as the Stop Grant state and greatly diminishes the processor's power consumption. In addition, this message is broadcast on the FSB to inform the chipset that the Stop Clock request has been honored.
07h	SMI Acknowledge. In response to a System Management interrupt received on SMI# (or via the APIC bus), the processor takes the following steps: • It ceases to execute the currently executing program. • Using the Special transaction, it broadcasts the SMI Acknowledge message. Note that the SMMEM# signal is asserted in packet B of this transaction to inform the Central Agent that until it indicates otherwise, System Management Memory is being addressed (rather than normal system memory). • The processor performs a series of memory writes to System Management Memory to save the contents of its register set. The processor is taking a "snapshot" of its context (i.e., register set contents) at the point of interruption. • The processor then fetches and executes the SMM handler routine. • The final instruction in the SMM handler is always the RSM (Resume) instruction. When it is executed, the processor performs a series of memory reads to reload its register set from the register save area in SMM. • After reloading its register set, the processor transmits a second SMI Acknowledge message, this time with SMMEM# deasserted, informing the Central Agent that it will no longer be addressing System Management Memory. • The processor then resumes execution of the interrupted program.
08h-FFh	Reserved.

The BTM Transaction Aids in Program Debug

The Problem

Before processors had internal caches, every memory read and write was performed on the FSB. A bus analyzer connected to the FSB could therefore see every instruction that was fetched from memory for decode and execution. It could see when a branch instruction was fetched and, when it was subsequently decoded and executed, it would see the processor alter its program flow when the processor began to issue memory read requests from the branch target address. In other words, the analyzer had full visibility of the program being executed.

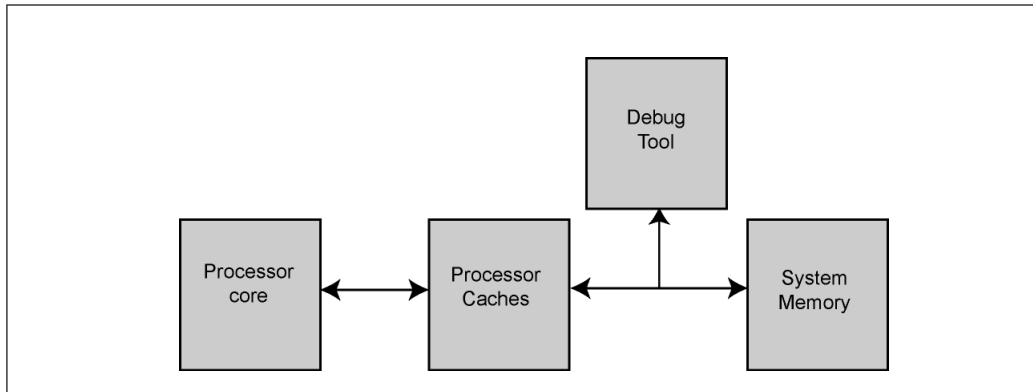
Refer to Figure 14-4 on page 279. When a processor incorporates an internal cache, however, program execution tracing can be a challenge. After the processor reads a program into the internal code cache, subsequent accesses to the same memory locations are not propagated through to the FSB. A FSB analyzer doesn't know what's going on. Specifically, it can't tell when the processor executes a branch instruction and, as a result, alters its program flow to a different part of the program that may already be in the code cache.

The Solution

The Pentium® and P6 processors can be forced to generate a message each time that it executes a branch and changes its program flow. The transaction type is the Branch Trace Message (BTM) transaction and it supplies the address of the branch instruction as well as the address of the instruction being branched to. This capability must be enabled by software, however.

Chapter 14: P6 FSB Central Agent Transactions

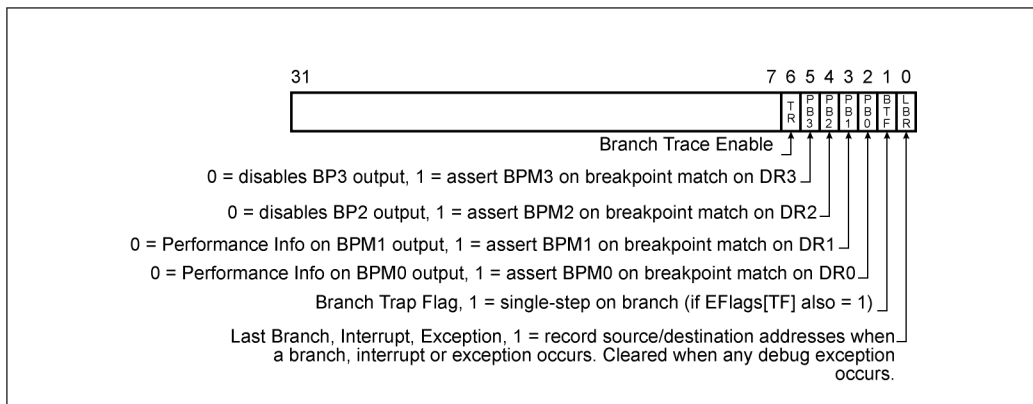
Figure 14-4: A Cache Can Hide Program Execution from an External FSB Analyzer



Enabling BTM Capability

The programmer enables or disables branch trace message transaction capability using a bit in an MSR (Model Specific Register) called the **DEBUGCTL** (Debug Control) register (see Figure 14-5 on page 279). Once enabled, the processor generates a BTM transaction each time that a branch executes and causes a change in program flow (in other words, the branch to the new address is taken). This is the only Central Agent transaction that outputs data on the data bus. It outputs information during the Data Phase to inform a debug tool which branch has been taken as well as the address that is being branched to.

Figure 14-5: **DEBUGCTL** MSR



Branch Trace Message Transaction

As stated earlier, Interrupt Acknowledge and BTM are the only Central Agent transactions that transfer data. The Interrupt Acknowledge transaction reads the vector from the 8259A Interrupt Controller, and the BTM transaction writes data to an external debug tool.

Proper Response to a BTM Transaction

The proper response is the No Data response (because it's a write). Intel® does not say that other responses are not permitted, so the author interprets this to mean that, although unlikely, the Retry, Deferred, and Hard Failure responses would also be permitted.

Detailed Description of the BTM Transaction

Refer to chapter 16 of the MindShare book entitled *Pentium® Pro and Pentium® II System Architecture*, Second Edition for a detailed description of the P6 BTM transaction.

15

Self-Modifying Code and the P6

Introduction

This chapter describes Self-Snooping as it is handled on the P6 processor family. Intel®'s Pentium® 4 documentation does not provide a detailed description of self-snooping, but this chapter describes an example problem and how self-snooping ensures proper program execution.

Description

Normally the instructions that comprise a program are always executed as they were originally written by the programmer. Sometimes, however, the programmer chooses to start off executing an instruction or series of instructions as originally composed, but then later modifies the instruction(s) on the fly so that when executed the next time, it will execute in its modified form and have a different effect. This is accomplished by reading the instruction(s) into the processor's register set, modifying it, and then writing it back to memory. The next time the processor fetches the modified instruction(s), they execute in their modified form and have the desired (i.e., altered) effect.

Generally speaking, Self-Modifying Code (SMC) is considered evil. Processor designers have to support it because people write it, but they usually don't optimize its performance; quite the opposite: you usually take quite a performance hit when it's executed. As one example, read the section entitled "The Internal Snoop" in MindShare's book entitled *Pentium® Processor System Architecture* (published by Addison-Wesley). For another, refer to the section on Self-Modifying Code in MindShare's book entitled *PowerPC System Architecture* (published by Addison-Wesley).

Refer to Figure 15-1 on page 284 through Figure 15-24 on page 307. The following numbered list defines the Pentium® Pro's method for handling Self-Modifying Code:

1. The first time that a piece of code is executed, the prefetcher issues a request for the line of code that contains the instructions (see Figure 15-2 on page 285).
2. If there is a code cache miss, the code cache issues the request to the L2 Cache (see Figure 15-3 on page 286 and Figure 15-4 on page 287).
3. If this also results in a miss, the L2 Cache issues a request to the FSB Interface Unit to fetch the line from memory (see Figure 15-5 on page 288).
4. When the line is returned from memory (see Figure 15-6 on page 289), it is placed in the L2 Cache (see Figure 15-7 on page 290) and is also placed in the code cache. The code cache, in turn, passes it on to the prefetch streaming buffer (what used to be called the prefetch queue).
5. The instructions are decoded into μ ops and are executed in their original form (see Figure 15-8 on page 291).
6. When the programmer later wishes to modify one or more instructions in this program, he or she uses a MOV instruction to read the instruction into a register (see Figure 15-9 on page 292).
7. The memory read request is submitted to the data cache, resulting in a miss (the instruction is in the code cache, not the data cache). The data cache issues a request to the L2 Cache, resulting in an L2 Cache hit (see Figure 15-10 on page 293).
8. The L2 supplies the data (it's really code being temporarily treated as data) to the L1 Data Cache and to the load execution unit which places the data in the target register (see Figure 15-11 on page 294).
9. The programmer then modifies the instruction in the register (see Figure 15-12 on page 295).
10. Using a MOV instruction, the programmer writes the data (i.e., the altered instruction) back to memory (see Figure 15-13 on page 296).
11. The write request is submitted to the L1 Data, L1 Code, and L2 Caches as well as to the Prefetch Streaming Buffer (see Figure 15-14 on page 297).
12. The code cache and L2 Cache both have hits and invalidate their copy of the code line.
13. As the code line may have already been read into the prefetch streaming buffer again (due to a branch prediction), the processor also performs a lookup in the streaming buffer. If it's a hit, the buffer is flushed. Note that whereas the lookup in the caches is based on the physical memory address, the streaming buffer lookup is based on the linear address (before the Paging Unit converts it to a physical address). This means that the programmer should always use the same linear address to fetch and to modify the instruction.
14. In case the line of code has already entered the instruction pipeline, the entire instruction pipeline is blown (see Figure 15-15 on page 298).
15. The Data Cache has a hit and updates its copy of the line. However, because

Chapter 15: Self-Modifying Code and the P6

it is also a hit on the Code Cache (the lookup is performed in all three caches simultaneously), the updated line is unloaded from the Data Cache (i.e., it's placed in a 32-byte writeback buffer, invalidated in the Data Cache, and is scheduled to be written back to memory). See Figure 15-16 on page 299.

16. The modified line is immediately unloaded from the L1 Data Cache and forwarded to a Writeback buffer within the FSB Interface Unit (see Figure 15-17 on page 300).
17. The line is then invalidated in the L1 Data Cache (see Figure 15-18 on page 301).
18. The next time that the prefetcher attempts to read the line (to execute the code again), it results in a miss on both the L1 Code and L2 Caches (see Figure 15-19 on page 302).
19. The line read request is forwarded to the FSB Interface Unit (see Figure 15-20 on page 303).
20. The processor initiates a memory read transaction to read the line from memory (see Figure 15-21 on page 304).
21. The processor snoops its own read transaction (*self-snooping!*) in its caches (it always does this, I just didn't mention it before) and in its Writeback buffer(s). There are two possible cases: the modified line that was unloaded from the Data Cache earlier has either already been written back to memory or it has not:
 - **Case A:** If the self-snoop doesn't result in a hit in its writeback buffers, the line has already been written back to memory and the processor reads the line from memory into its L2 and Code caches.
 - **Case B:** If there is a hit on its writeback buffers (see Figure 15-22 on page 305), the line has not yet been written back to memory. The processor drives the modified line onto the FSB during the Data Phase of the transaction and reads it back into the L2 and Code Caches *at the same time!* System memory also receives the modified line (see Figure 15-23 on page 306).
22. The Code Cache passes the line to the prefetch streaming buffer and the instructions are decoded and executed in their modified form (see Figure 15-24 on page 307).

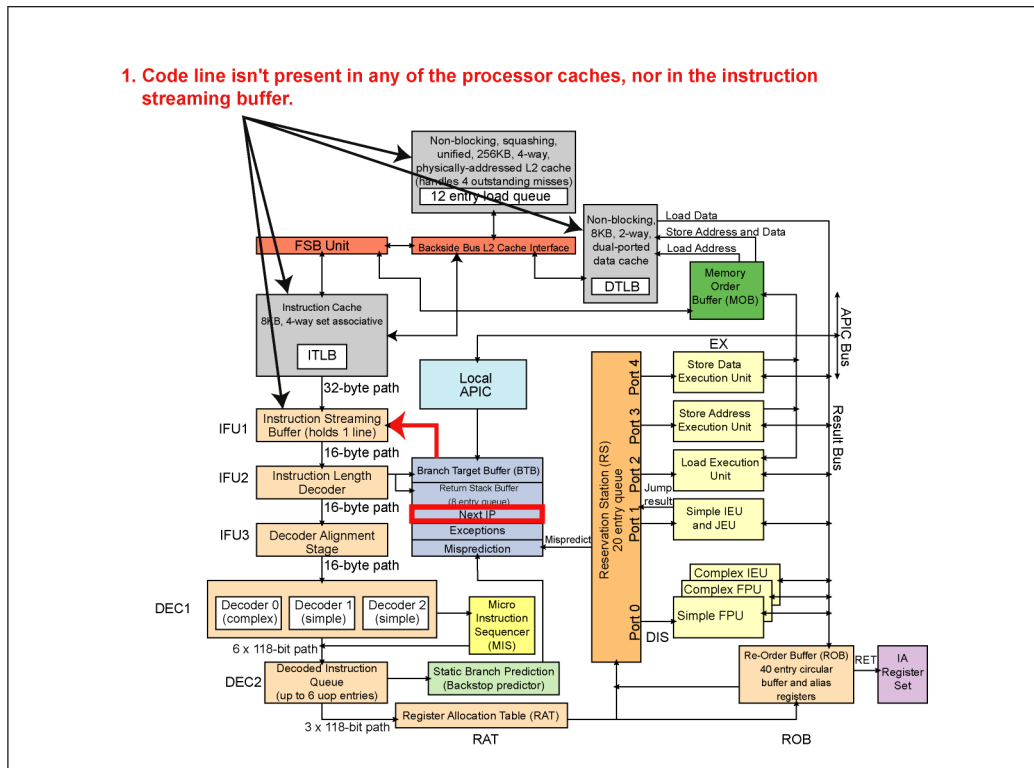
Don't Let Your Data and Code Get Too Friendly!

Refer to Figure 15-25 on page 308. Even though you may not write Self-Modifying Code, this scenario can still occur, resulting in terrible performance for your program. Consider the case where you place some data in memory immediately following a program. The instructions at the end of your program and the data may reside within the same line. When you read any of these data items into a

The Unabridged Pentium® 4

register, the entire line, comprised of mixed code and data, is read into the Data Cache. If you change the data in the register and then write it back to memory, it results in a hit on the Data, Code and L2 Caches. The procedure described in the preceding section is then performed (in other words, the processor thinks that you are modifying code in the Data Cache).

Figure 15-1: The Assumption



Chapter 15: Self-Modifying Code and the P6

Figure 15-2: The Code Line Is Requested

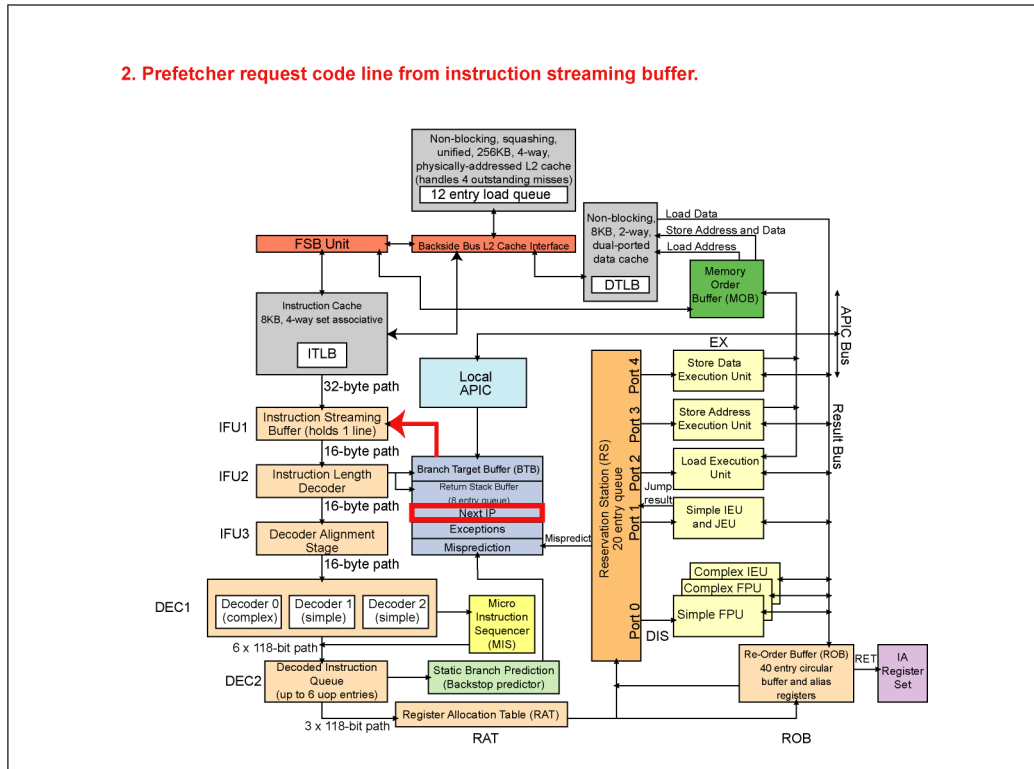
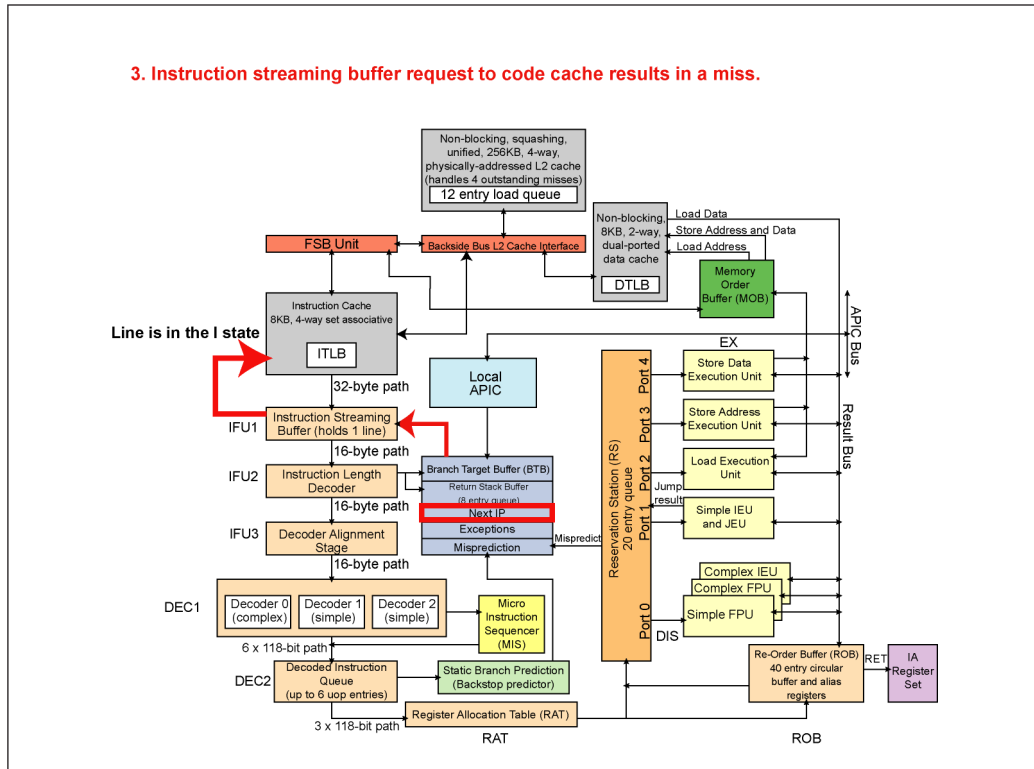
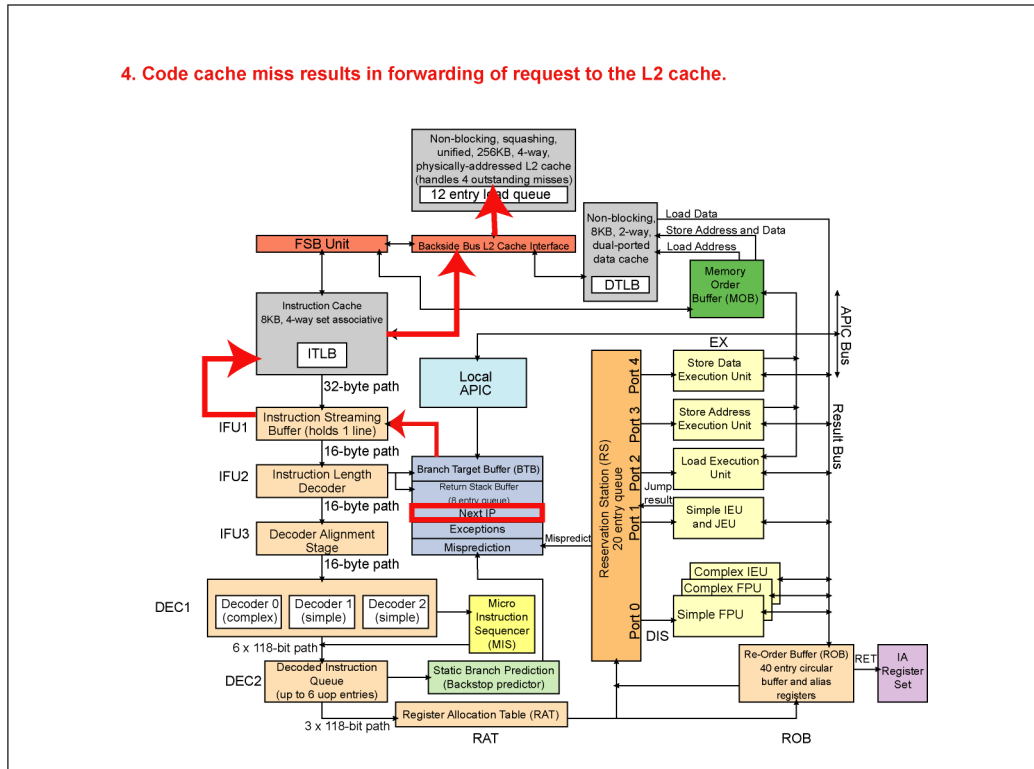


Figure 15-3: The L1 Code Cache Miss



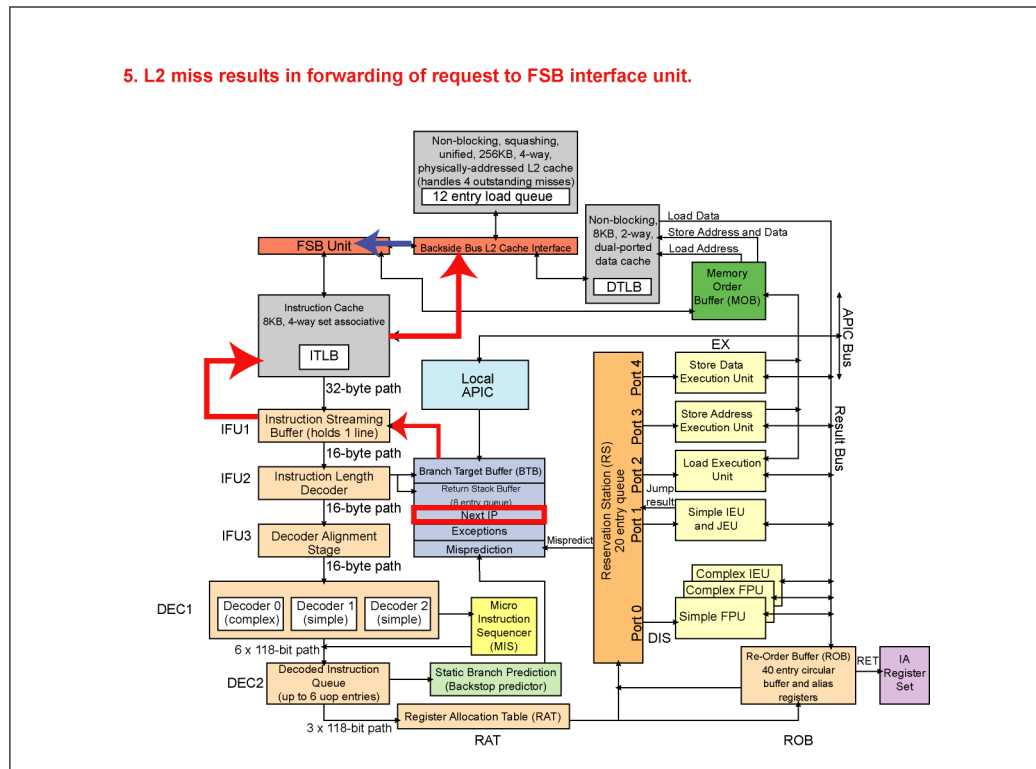
Chapter 15: Self-Modifying Code and the P6

Figure 15-4: The Code Request Is Forwarded to the L2 Cache



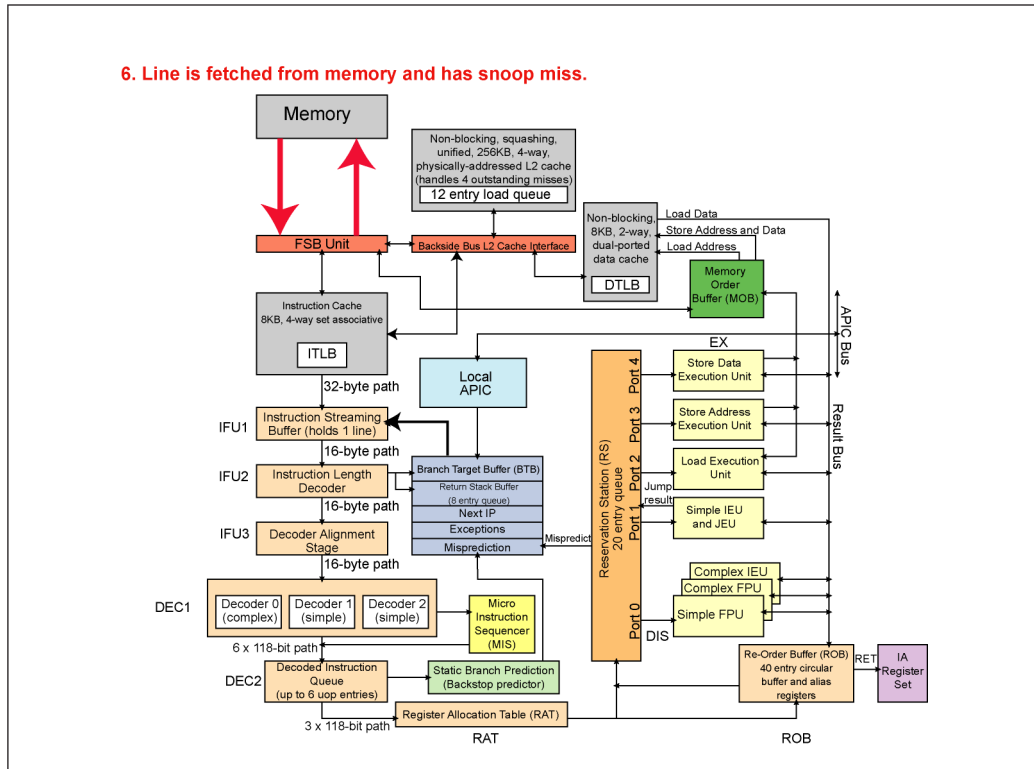
The Unabridged Pentium® 4

Figure 15-5: The Code Request Is Forwarded to the FSB



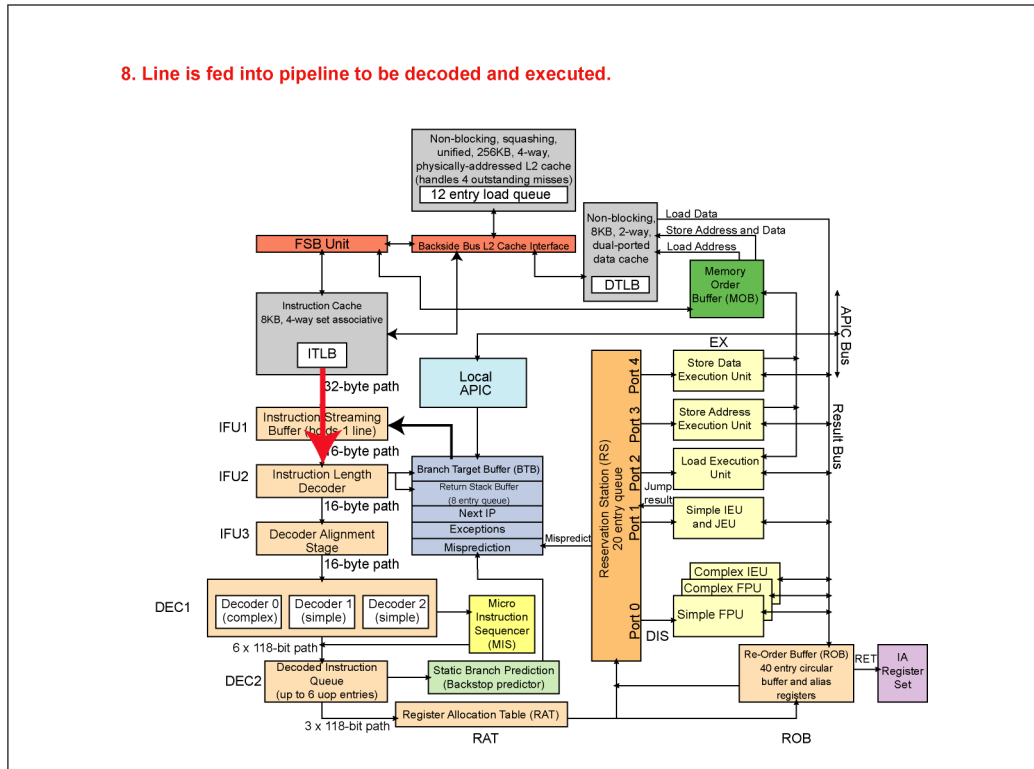
Chapter 15: Self-Modifying Code and the P6

Figure 15-6: The Line Is Fetched from Memory



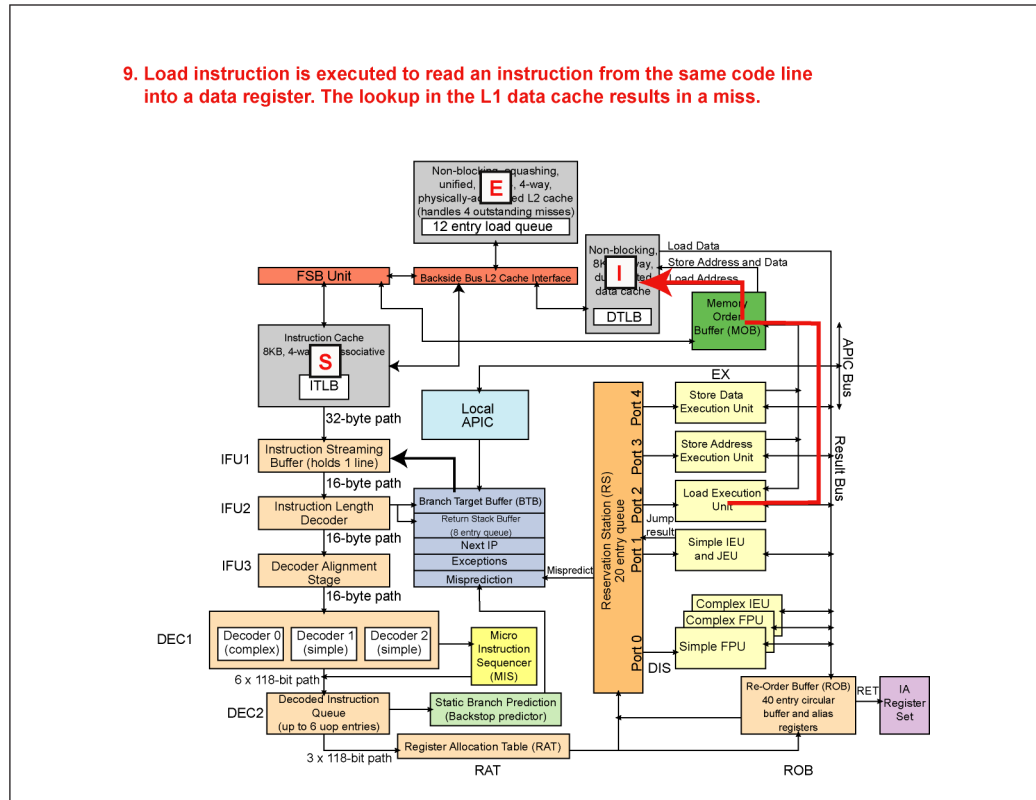
Chapter 15: Self-Modifying Code and the P6

Figure 15-8: The Line Is Fed Into the Pipeline



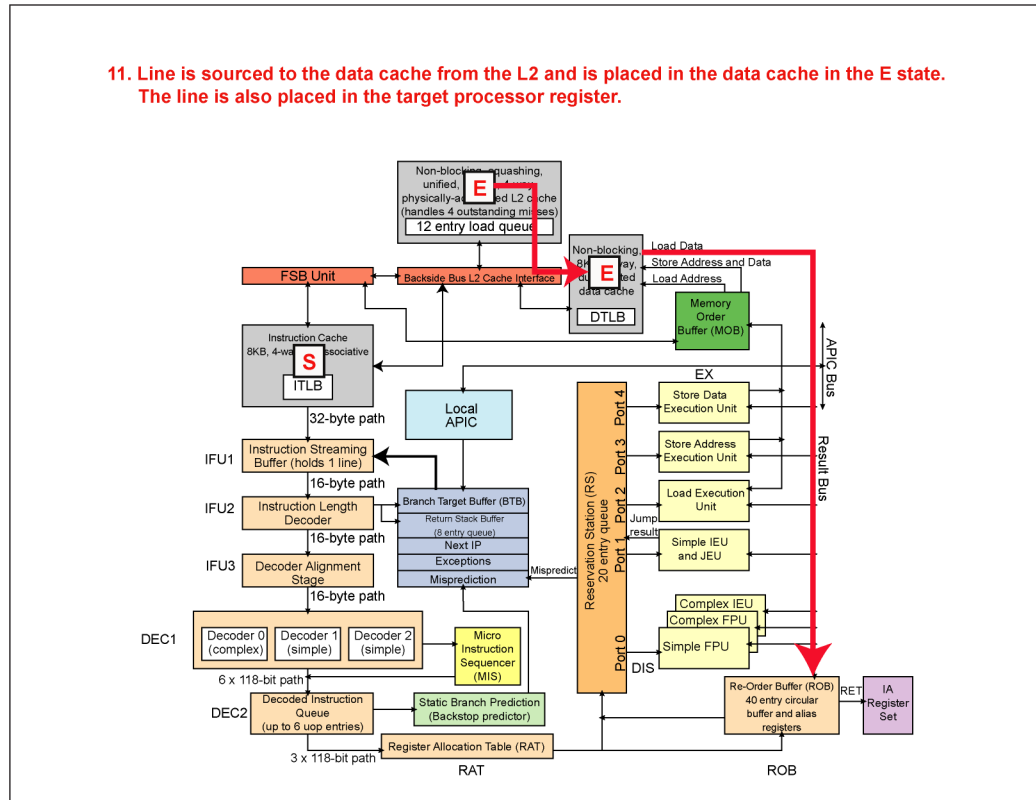
The Unabridged Pentium® 4

Figure 15-9: A Load Is Executed to Read an Instruction from Memory into a Register



The Unabridged Pentium® 4

Figure 15-11: The Line Is Forwarded to the L1 Data Cache and to the Register



Chapter 15: Self-Modifying Code and the P6

Figure 15-12: The Instruction Is Altered in the Register

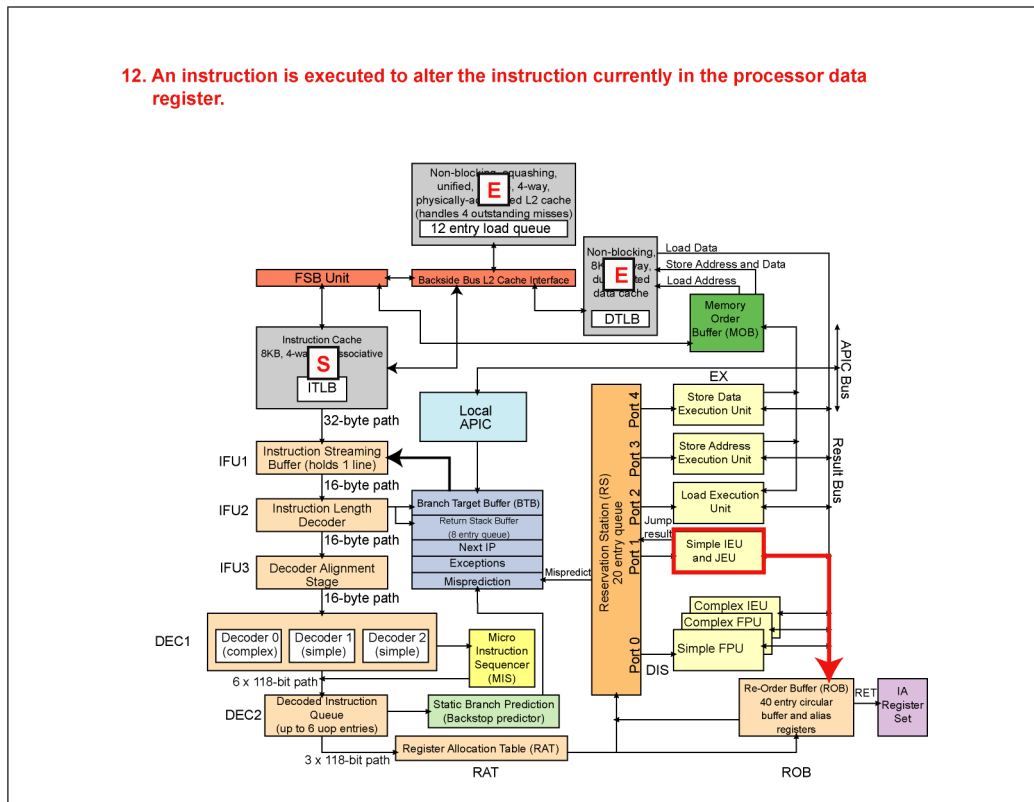
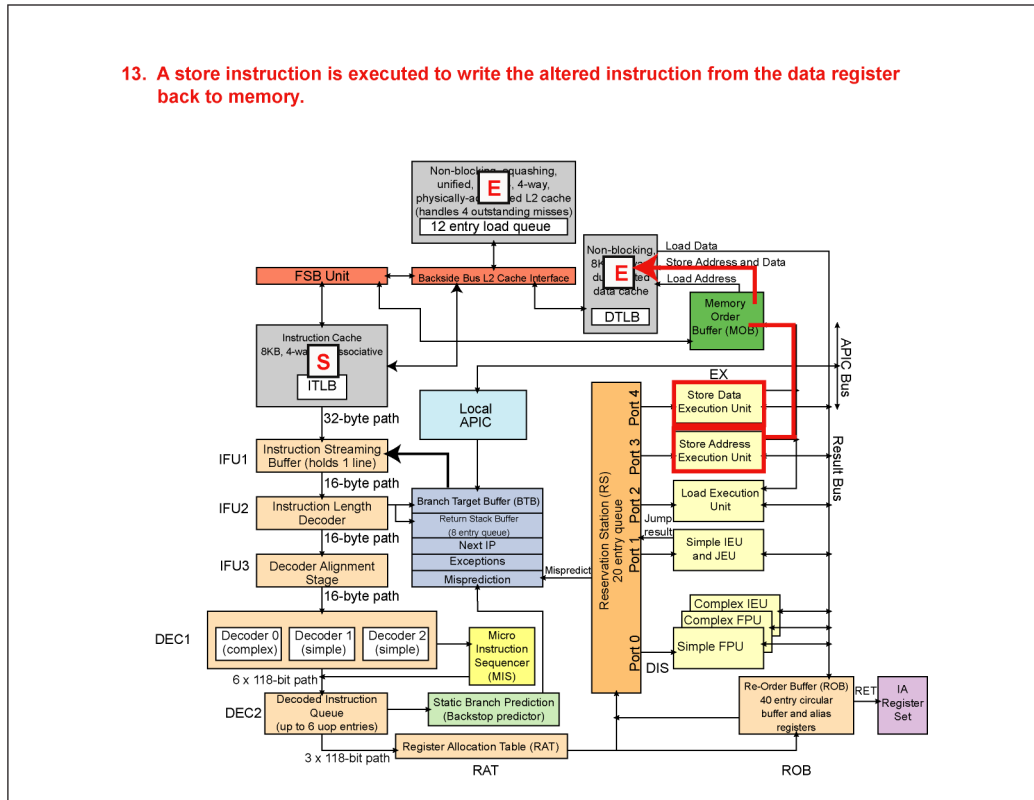


Figure 15-13: The Modified Instruction is Stored Back to Memory

13. A store instruction is executed to write the altered instruction from the data register back to memory.



Chapter 15: Self-Modifying Code and the P6

Figure 15-14: The Triple Lookup

14. When a store hits on the L1 data cache, it is also snooped in the L1 code cache and the L2 cache. On a triple hit, the line is invalidated in the L2 cache and L1 code cache.

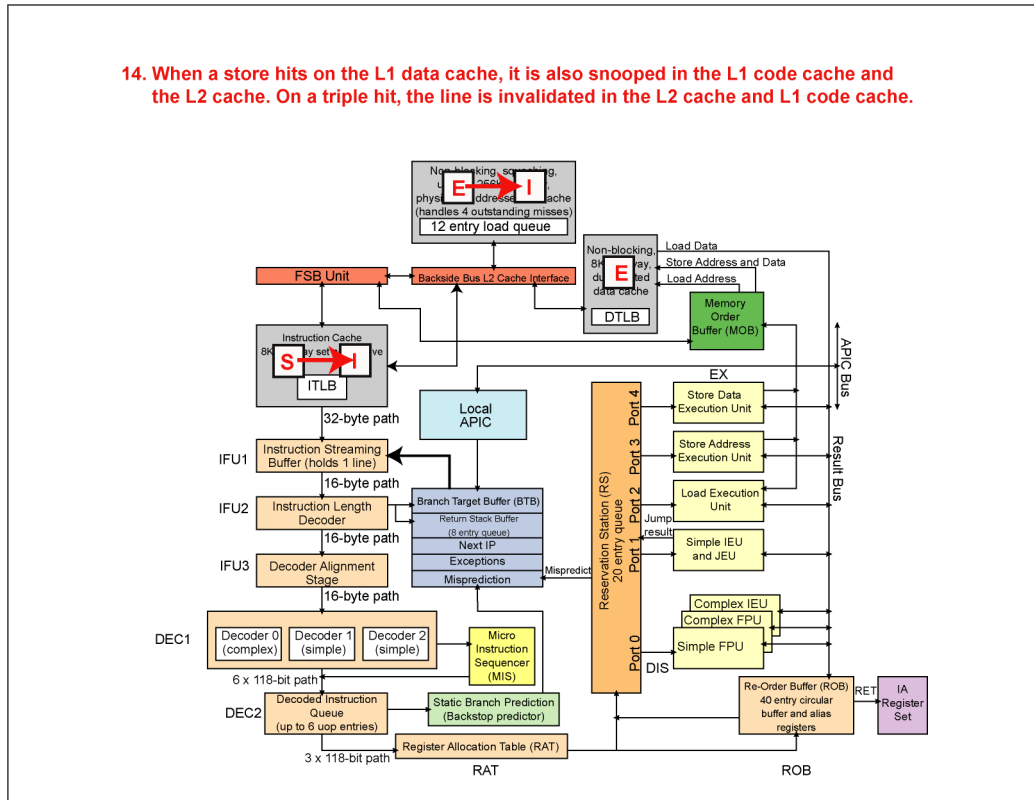
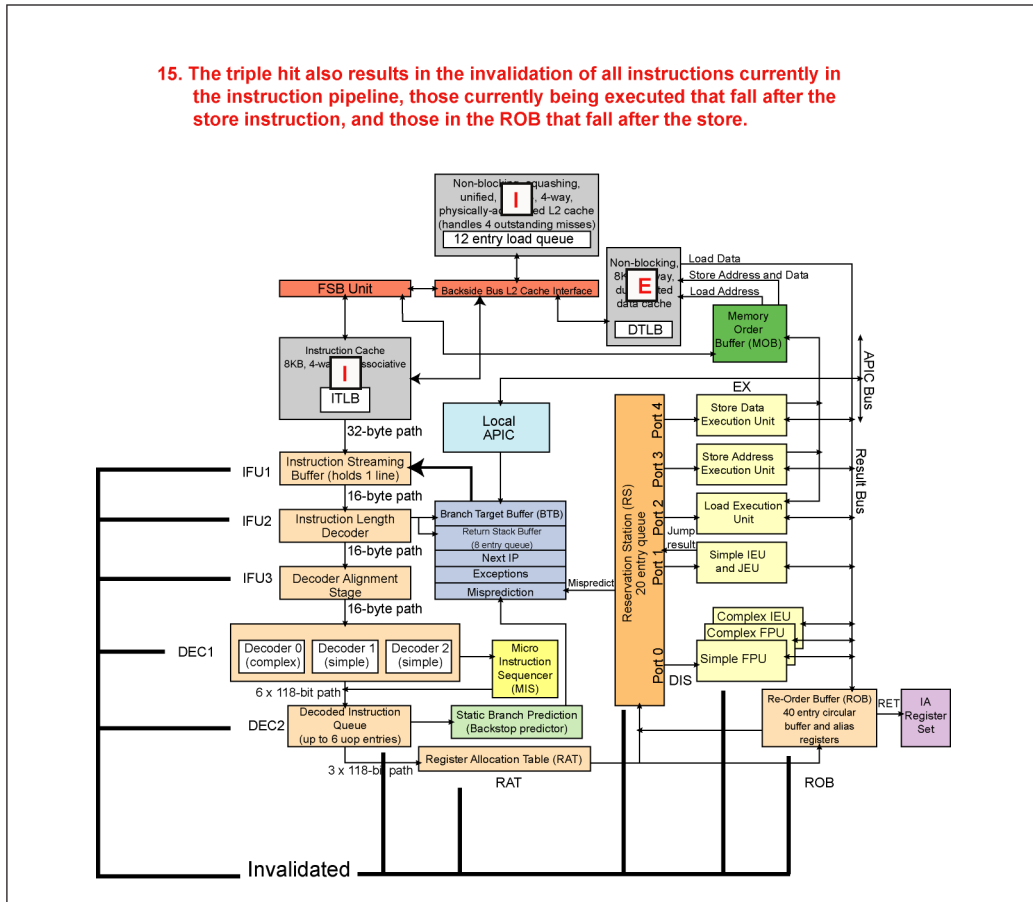


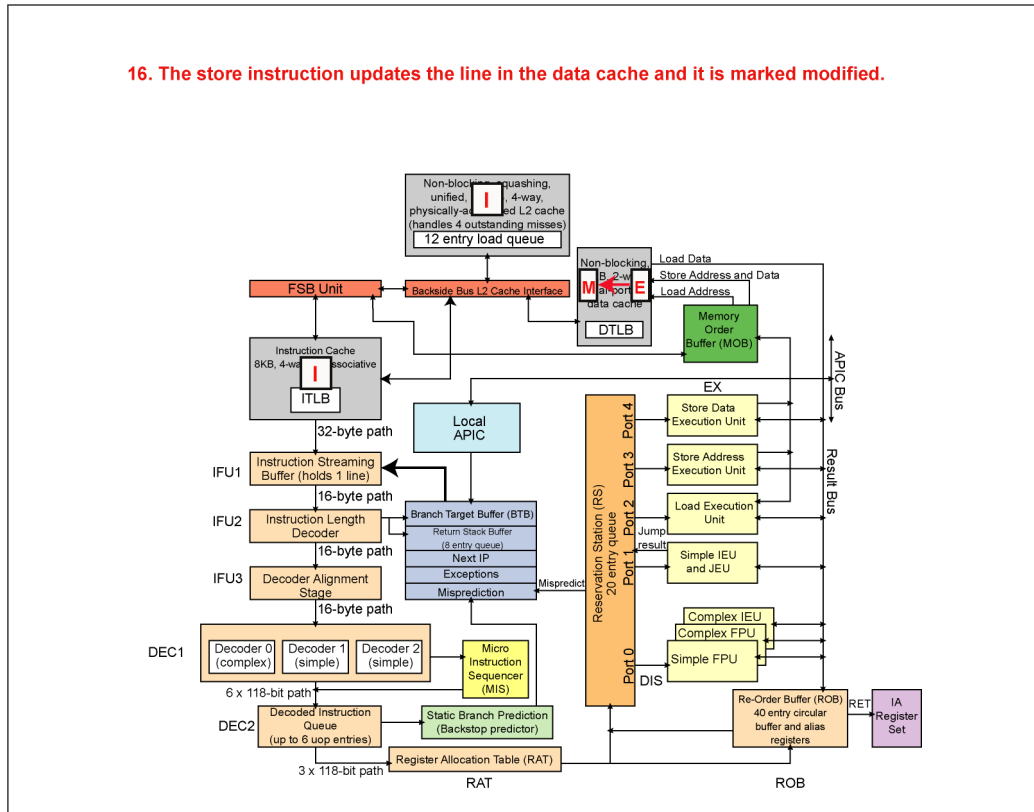
Figure 15-15: The Pipeline Is Blown



Chapter 15: Self-Modifying Code and the P6

Figure 15-16: The L1 Data Cache Line Is Updated

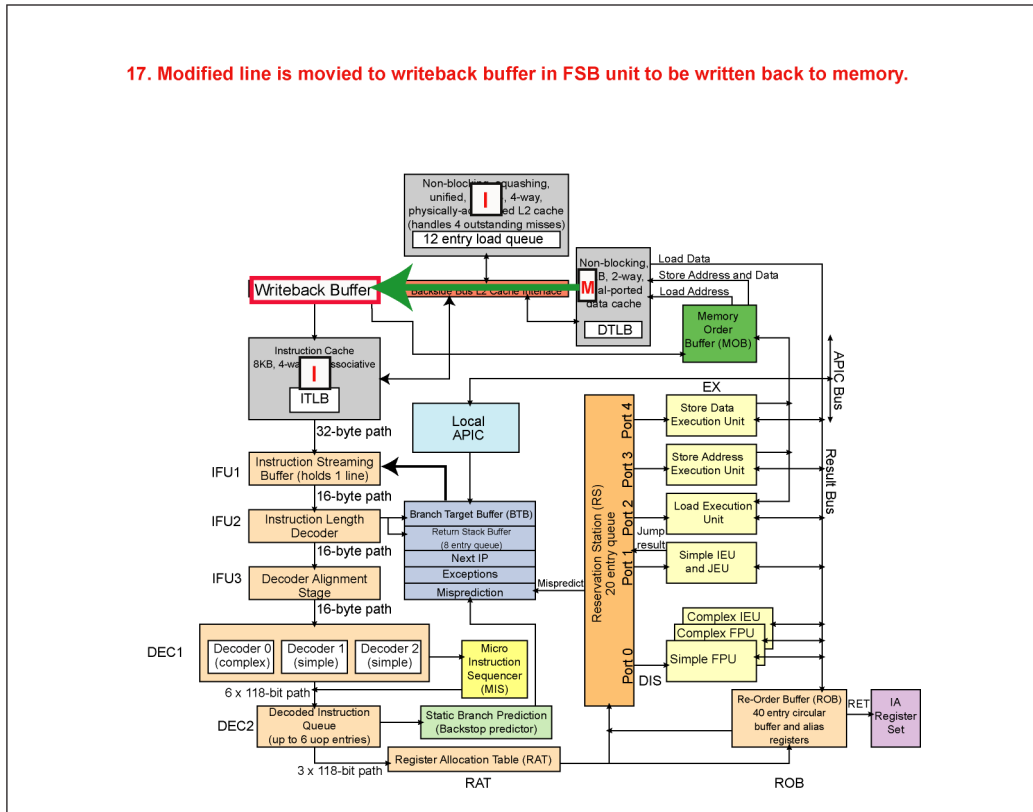
16. The store instruction updates the line in the data cache and it is marked modified.



The Unabridged Pentium® 4

Figure 15-17: The Modified Line Is Forwarded to the FSB Interface Unit

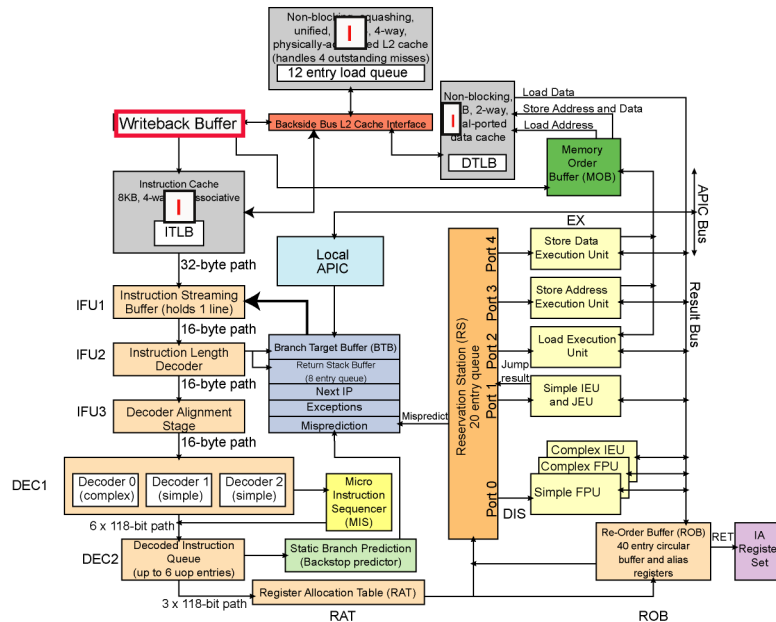
17. Modified line is moved to writeback buffer in FSB unit to be written back to memory.



The Unabridged Pentium® 4

Figure 15-19: The Next Time the Line Is Fetched, It Results In a Cache Miss

19. The next time the core attempts to fetch that code line, it results in a miss on the code cache and the L2 cache. There are two possible cases:
- The modified line has already been written back to memory.
 - The modified line is still in the processor's writeback buffer.
- This example assumes that the line still resides in the writeback buffer (case b).



Chapter 15: Self-Modifying Code and the P6

Figure 15-20: The Line Read Request Is Forwarded to the FSB Interface Unit

20. The instruction prefetcher issues a request for the line, resulting in:
- a miss on the instruction streaming buffer.
 - a miss on the code cache.
 - a miss on the L2 cache.
- The line request is issued to the FSB unit to be read from memory.

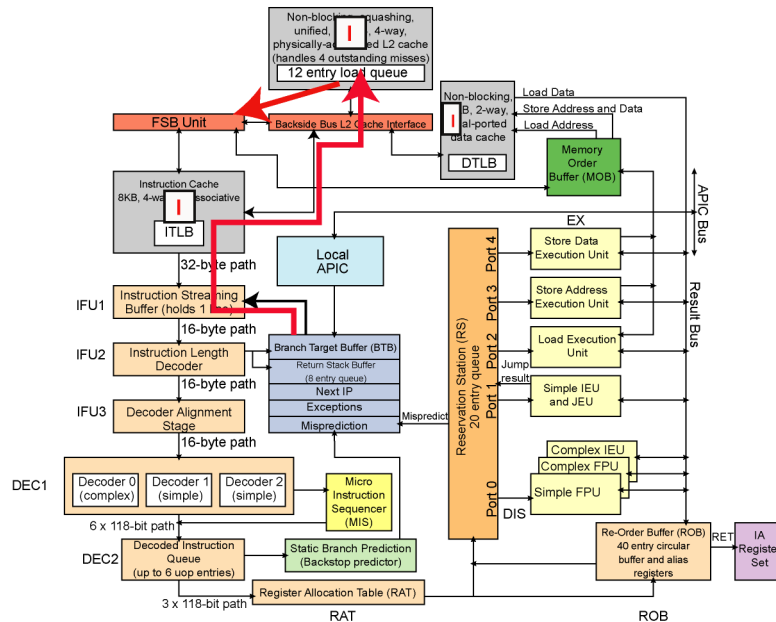
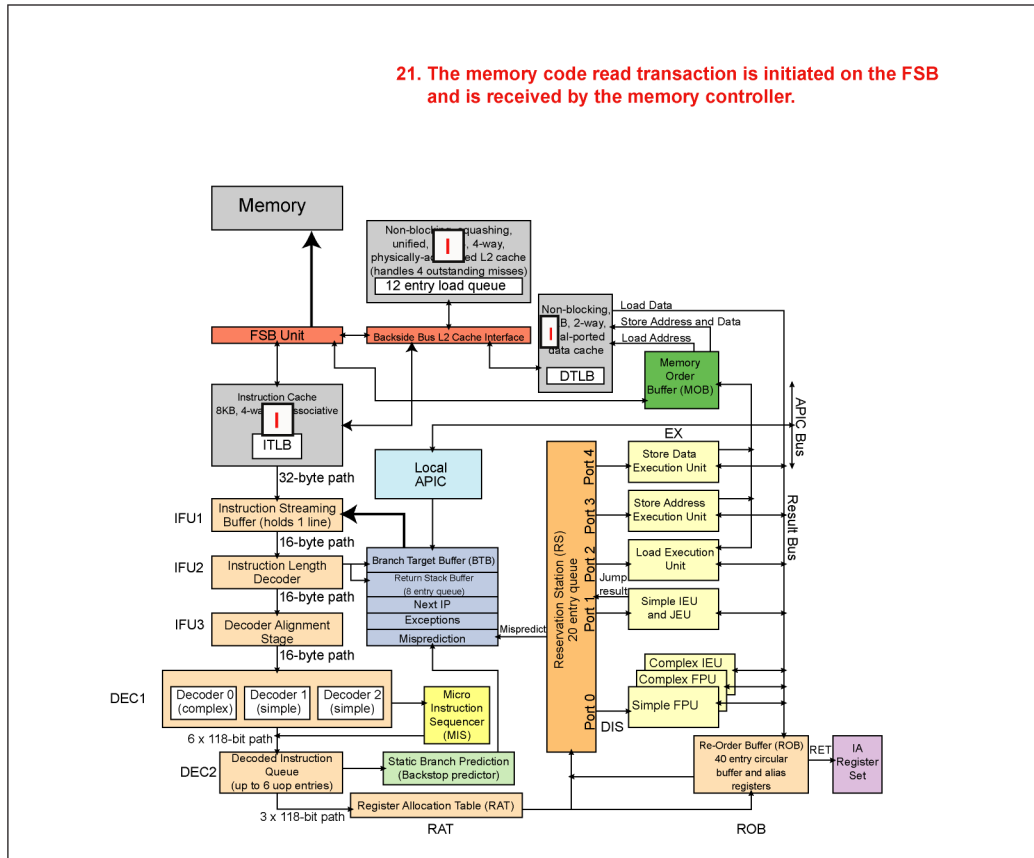
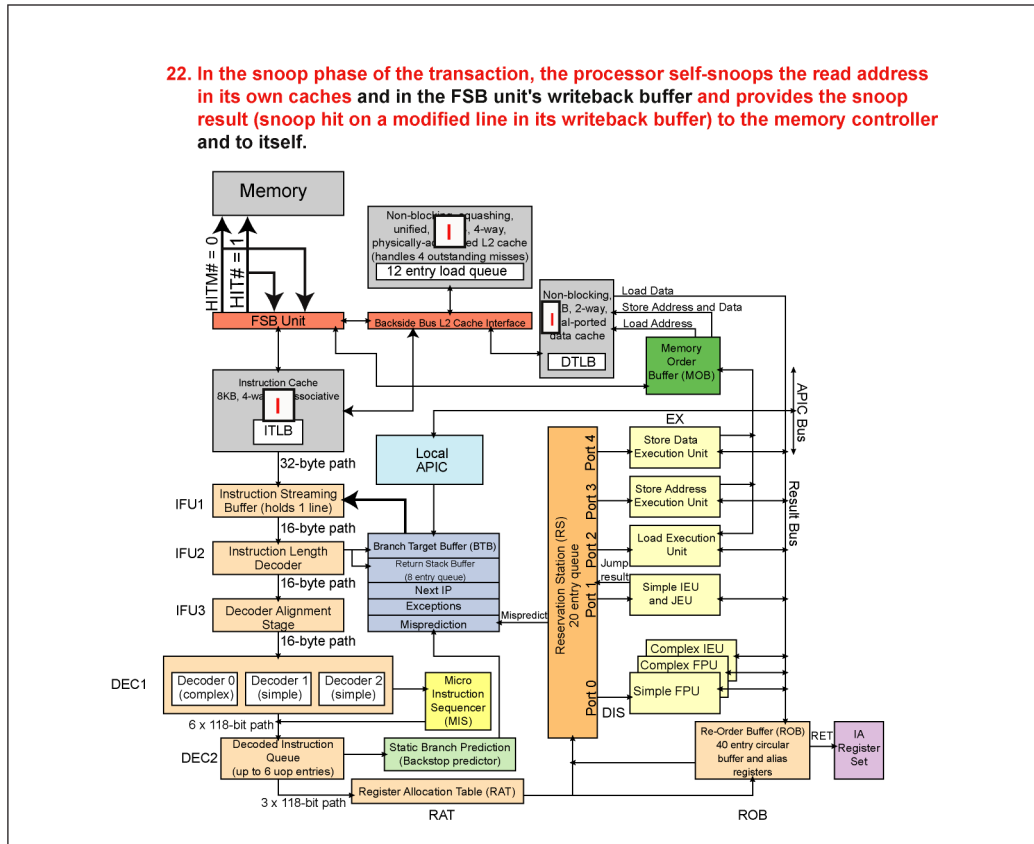


Figure 15-21: The Memory Code Read Transaction Is Performed on the FSB



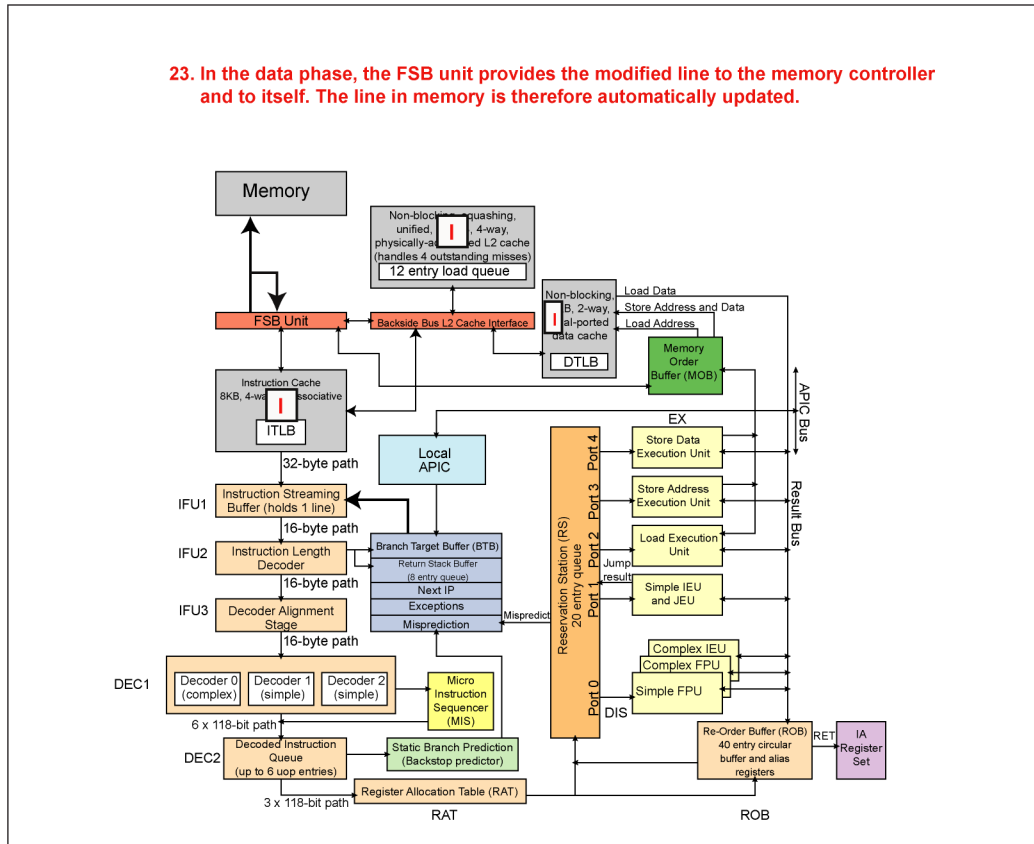
Chapter 15: Self-Modifying Code and the P6

Figure 15-22: The Self-Snoop Results in a Hit on the Modified Line in the Writeback Buffer



The Unabridged Pentium® 4

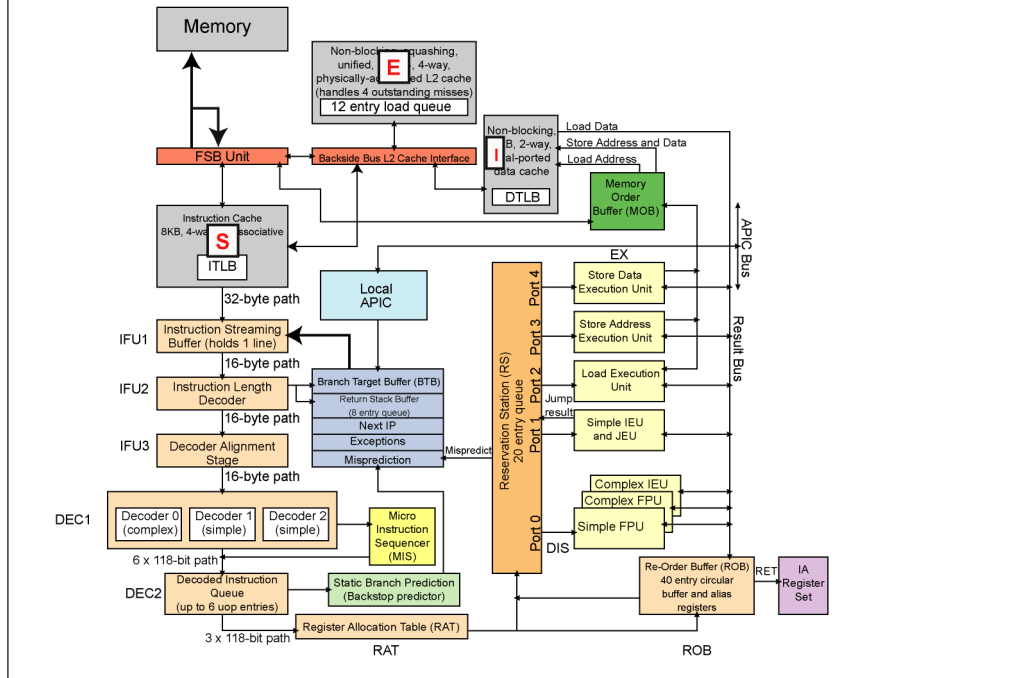
Figure 15-23: The FSB Interface Unit Provides the Modified Line to Memory and to the CPU



The Unabridged Pentium® 4

Figure 15-25: Don't Let Your Code and Data Get Too Friendly

Don't let your code and data get too friendly! If code and data co-resides in the same line and your program performs any updates to a data item in the line, the processor will interpret this as self modifying code and the result is terrible performance.



16 *Pentium® 4 FSB Screenshots*

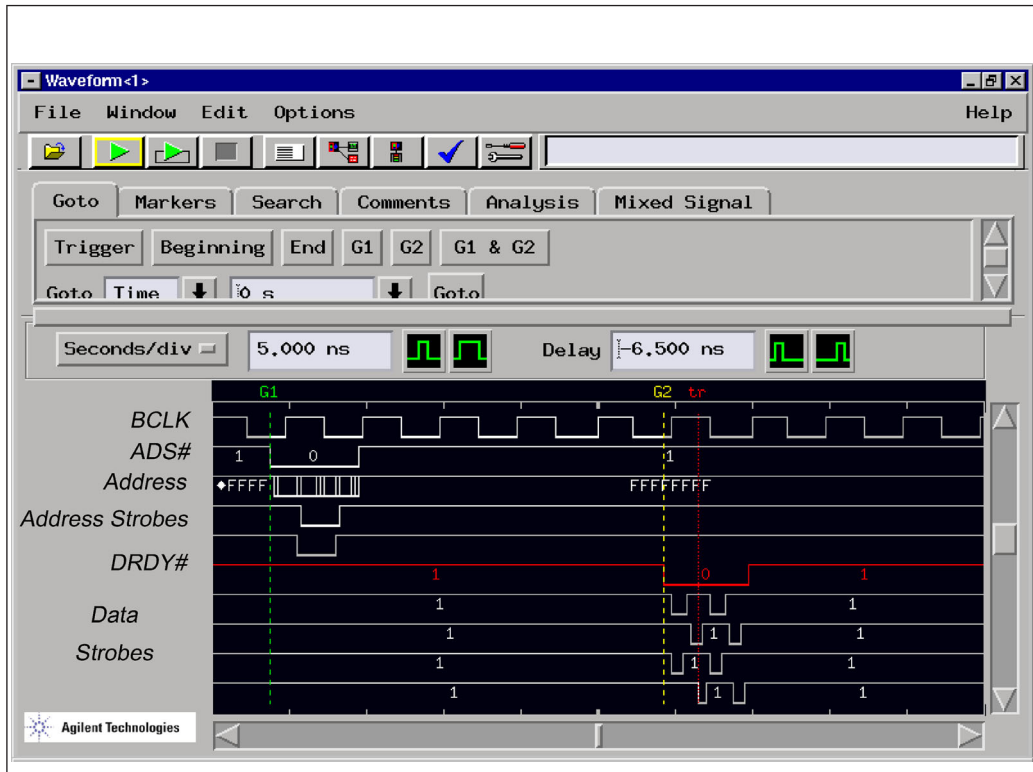
Courtesy of Agilent Technologies

The author would like to thank Agilent Technologies for the use of a Pentium 4 FSB analyzer. The example screenshots of FSB transactions in this chapter were captured using the analyzer.

Example 1

This is an example of a 64-byte memory write transaction (the delivery of the response is not shown).

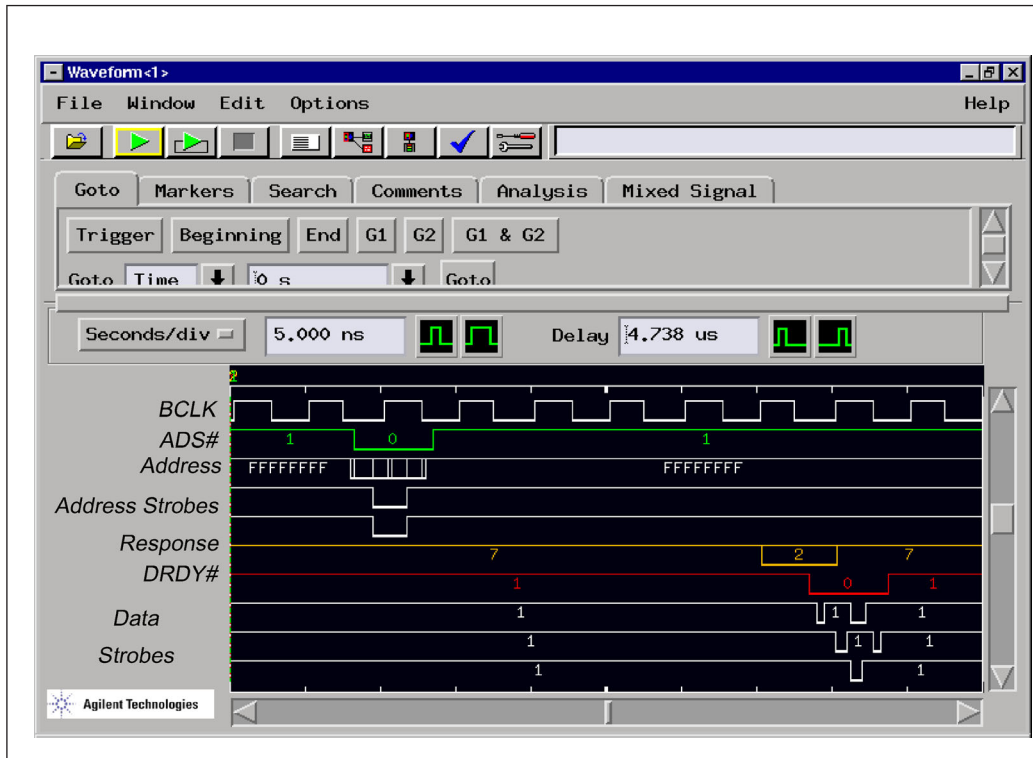
Figure 16-1: Example 1



Example 2

This is another example of a 64-byte memory write transaction. In this case, the delivery of the No Data response is shown.

Figure 16-2: Example 2

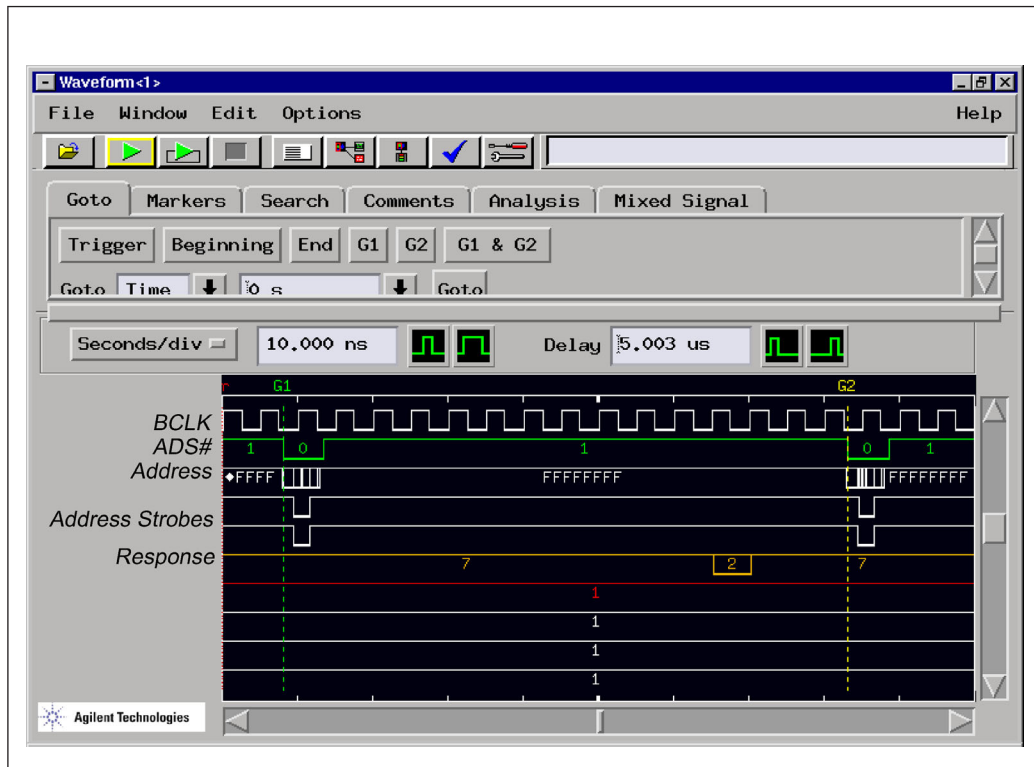


Example 3

This is an example of a transaction that doesn't transfer any data. It could be any of the following:

- A 0-byte MRI issued by the processor to kill Shared lines in the caches of other processors.
- A 0-byte MRI issued by the MCH as a snoop transaction (because a PCI master is performing a write to system memory).
- A 0-byte Memory Read issued by the MCH as a snoop transaction (because a PCI master is performing a read from system memory).
- A Special transaction generated by the processor to broadcast a message.

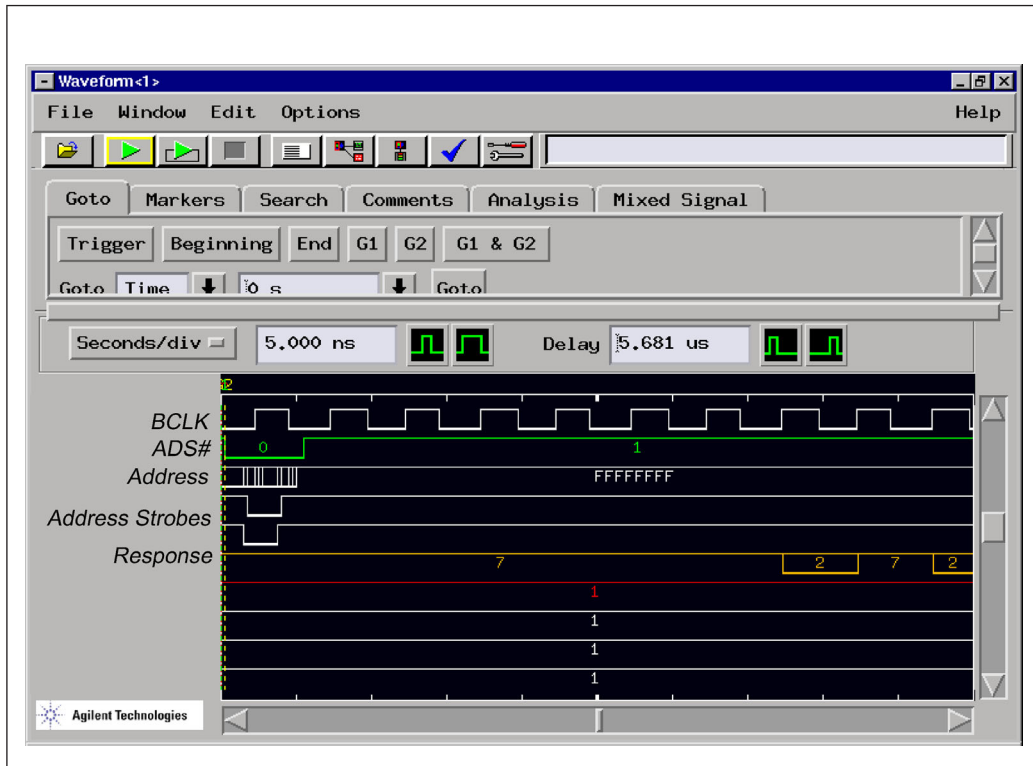
Figure 16-3: Example 3



Example 4

This example shows the response for the previous transaction followed immediately by the response for the current transaction.

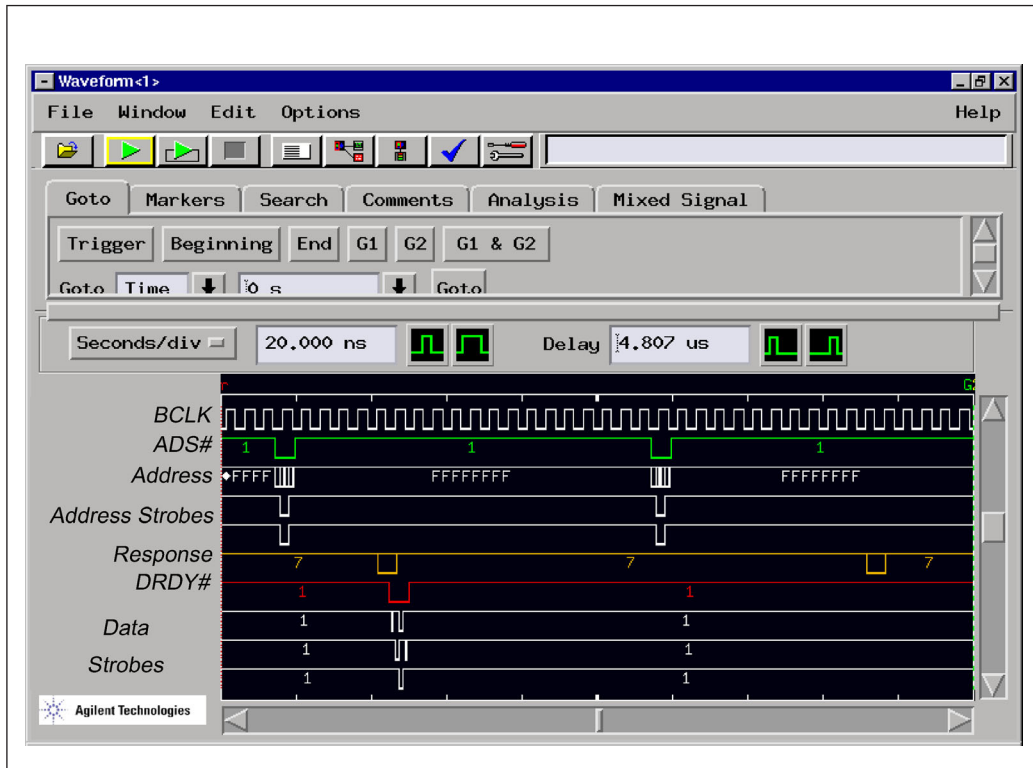
Figure 16-4: Example 4



Example 5

This example shows a memory write transaction followed by a transaction that doesn't transfer any data.

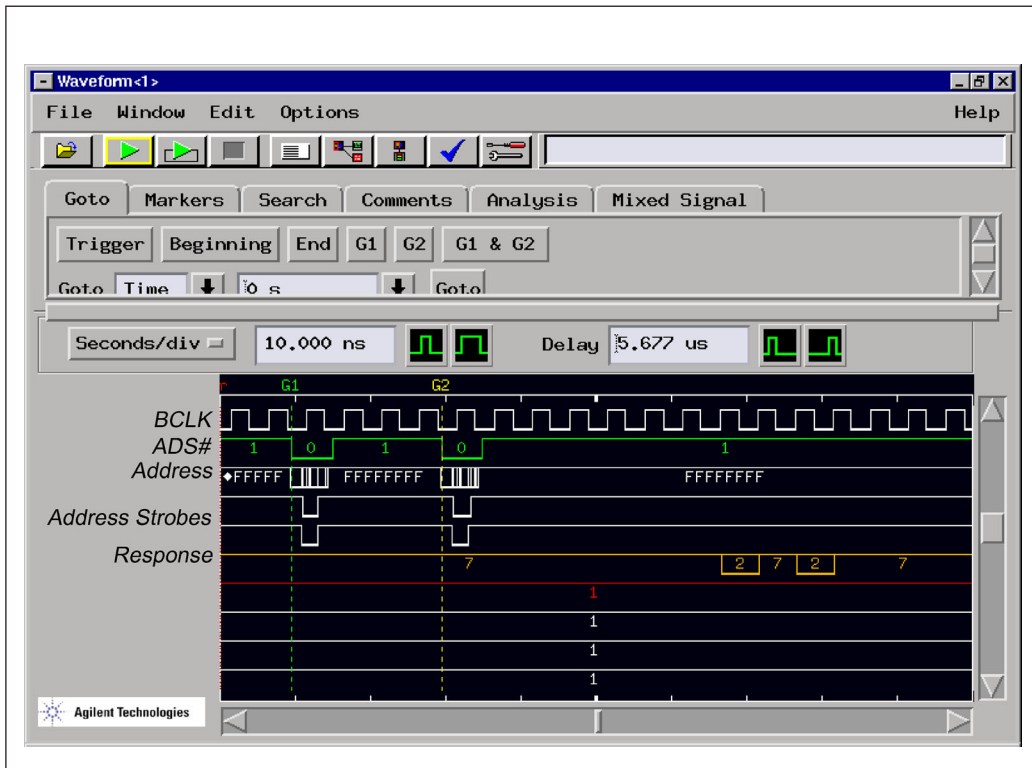
Figure 16-5: Example 5



Example 6

This example shows two transactions neither of which transfer any data (e.g., two back-to-back snoop transactions generated by the MCH).

Figure 16-6: Example 6



Symbols

μ ops 174

Numerics

486 Branch Handling 181
8259A interrupt controller 273

A

A[35:3]# 237
ACPI Table 191
Address Bus Width, P6 147
Address Bus Width, Pentium II 153
Address Strobe 237
ADS# 237
AERR# 243
Agent ID 196, 218
Agents, FSB 193
AGTL+ 205
Alignment, Data 189
analyzer 278
AP[1:0]# 237
APIC, Local 172
Arbitration Algorithm, P6 218
ATTR[7:0]# 238
Attribute signals 238
AutoHalt Power Down State 148

B

Backside Bus (BSB) Interface Unit 171
Backstop mechanism 187
BE[7:0]# 238
Block Next Request 229
Blocking New Requests 229
BNR# 229
BPRI# 223
Branch Prediction 181
Branch Prediction, P6 dynamic 184
Branch Prediction, Pentium 181
Branch Trace Message Transaction 170,
241, 272, 278
BSB Interface Unit 171
BSB Speed and Width, P6 145

BSB Speed and Width, Pentium II 153
BSP Selection, P6 191
BTB 182, 184
BTM 170
BTM Capability on the P6, enabling 279
BTM transaction 272, 278
Bus Parking 221
Busy / Idle State 220
Byte Enables 238, 275

C

Cartridge 151
Celeron 161, 162
Central Agent 271
Central Agent Transactions, P6 271
Context Switches 190
Coppermine 164
Covington 161

D

D[63:0]# 258, 263
Data Bus 258, 263
Data Bus Busy 257, 263
Data Bus ECC / Parity 263
Data Bus Inversion 258
Data Phase 257, 263
Data Phase Signal Group, P6 263
Data Phase(s) 201
Data Ready 258, 263
Data Strobe Signals 258
DBSY# 257, 263
DEC1 179
DEC2 179
Deep Sleep State 155
Defer Enable 265
DEFER# 246, 266
deferral 195
Deferral Basics, P6 265
Deferred 261
Deferred ID 238
Deferred Reply Transaction 170, 196, 241,
268

Index

Deferred Response 195

DEN# 265

DEP[7:0]# 263

Deschutes 143, 159

DIBA 150

DID[7:0]# 238

differential receiver 206

DIS 180

Dixon 162

DRDY# 258, 263

Dual Independent Bus Architecture 150

Dual-Cavity PGA Package 144

E

ECC 258

Error Phase 200, 233, 234, 243

EX 180

EXF[4:0]# 239

Extended function signals 239

F

false register dependency 175

Flush 276

Flush Acknowledge 277

FLUSH# 277

FSB Arbitration Basics, P6 215

FSB Arbitration Scheme, P6 147

FSB Arbitration Scheme, Pentium II 153

FSB Basics, P6 193

FSB Electrical Overview, P6 205

FSB Interface Unit, P6 167

FSB is a Synchronous Bus 211

FSB Protocol, P6 147

FSB Request & Error Phase Basics, P6 233

FSB signal groups, P6 198

FSB Speed, Pentium II 153

FSB Speed, Pentium Pro 146

G

GTL 206

GTL+ 206

Gunning Transceiver Logic 206

H

Halt 276

Halt/Grant Snoop State 150

Hard Failure response 261

HIT# and HITM# 246

HLT instruction 276

I

IA32 Data Register Set 174

IA32 instructions 173

Idle 260

IFU1 179

IFU2 179

IFU3 179

Implicit Writeback 261

In Order Queue (IOQ) 204

Instruction Pipeline, P6 178

Interrupt Acknowledge transaction 170,
241, 272, 273

INTR signal 273

INVD 276

IO Read transaction 241

IO Write transaction 241

IOQ 204, 229

J

JEU 182

Jump Execution Unit 182

K

Katmai 143, 163

Klamath 143

L

L1 Code Cache, P6 171

L1 Code Cache, Pentium II 153

L1 Code Cache, Pentium Pro 146

L1 Data Cache, P6 171

L1 Data Cache, Pentium II 153

L1 Data Cache, Pentium Pro 146

L2 Cache, Pentium II 152

L2 Cache, Pentium Pro 145

Legacy Code Execution Speed,
Pentium II 156
LOCK# 228
Locking 226
low-voltage swing 206
LVS 206

M

MCERR# 244
Memory Code Read transaction 242
Memory Data Read Transaction 242
Memory Line WriteBack transaction 169,
242
Memory Read and Invalidate
transaction 169, 242
memory semaphore 227
Memory Write 169
Memory Write (may be retried)
transaction 242
Mendocino 161
Message 170, 275
Mispredicted Branches 182
MMX 155
MP Table 191
MRI 169
Multiprocessing Table 191

N

No Data Response 261
NOP 276
Normal Data Response 262

O

OOO 174
Optimizations, P6 Code 188
out-of-order 174

P

P6 Processor Core 167
P6 Processor Family 143
Packet A Content 238
Packet B Content 238

packets A and B 237
packets, Request 233
PAE-36 Mode 147
parity 258
Parity, P6 Request Phase 234
Partial Stalls 190
PCI-to-ISA Bridge 273
Pentium Branch Prediction 181
Pentium II Overdrive 159
Pentium II, later models of 160
Pentium III Coppermine 164
Pentium III, the early 164
Pentium Pro 144
Posted Write Buffer 226
Power Conservation States,
Pentium II 154
Power Conservation States,
Pentium Pro 148
Priority Agent Arbitration, P6 223
Priority Agent Parking 225
Priority Request Agents 196
Processor Core, P6 171

R

RAT 176, 180
register alias table 176
ReOrder Buffer 174
REQ[4:0]# 237
Request & Error Phase Basics, P6 233
Request Agent 193, 196, 247
Request packets 233
Request Phase 200, 234
Request Phase Parity, P6 234
Request Phase signal group 198, 236
Response Agent 193, 247
Response Bus 259
Response Phase 201, 257
Response Phase parity 259
Response Phase Signal Group 259
Response Types 258
RET 186
RET1 180

Index

RET2 180

Retry Response 195, 260

Return Stack Buffer (RSB) 185

RMW 228, 242

ROB 174, 180

Rotating ID 218

RP# 237

RS[2:0]# 259

RSB 185

RSP# 259

S

semaphore 227

Serializing Instructions 189

Shared Resource 226

Shutdown 276

Sleep State 154

SMI Acknowledge 277

SMI# 277

Snoop Agents 193, 246

Snoop hit 194

Snoop hit on a modified line 194

Snoop miss 194

Snoop Phase 200

Snoop Phase duration 248

Snoop Phase Overview, P6 FSB 245

Snoop Phase, non-memory transactions
and the 255

Snoop Result 249

Snoop Stall 251

South Bridge 273

Special Transaction 170, 241, 272, 275

Stall, Snoop 251

Stalls, Partial 190

Static Branch Prediction, P6 187, 189

Stop Clock 277

Stop Grant Acknowledge message 277

Stop Grant State 149

STPCLK# 277

Symmetric Agent Arbitration 216

Symmetric Request Agents 196

Sync 277

T

Target Ready 259

Transaction Deferral 195

Transaction Phases 197

Transaction Tracking 203

TRDY# 259

U

Unified L2 Cache, P6 171

W

WBINVD 277

X

Xeon Cascades 165

Xeon Tanner 165

Xeon, Pentium II 162